



版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

介绍分布式系统基础理论，分析分布式系统中常用的主流技术分享实战案例，做到理论与实践相结合。

Broadview
www.broadview.com.cn

非卖品，仅供非商业用途或交流学习使用

分布式系统
常用技术及案例分析
(第2版)

柳伟卫◎著

中国工信出版集团 电子工业出版社

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

非卖品，仅供非商业用途或交流学习使用

柳伟卫

柳伟卫，网名waylau，老卫，80后程序员，关注编程、系统架构、性能优化。在IT公司担任过项目经理、架构师、高级开发顾问等职位，拥有多年Java开发经验，具有丰富的软件开发管理及系统架构经验。主导过多个大型分布式系统的设计与研发，参与过面向全球的供应链系统改造。在实际工作中，积累了大量的分布式系统、微服务架构经验。CSDN、开源中国、云栖社区等技术社区专家。

博客：https://waylau.com
邮箱：waylau521@gmail.com
微博：http://weibo.com/waylau521
开源：https://github.com/waylau

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用



目 录

第 1 章 分布式系统基础知识.....	1
1.1 概述	2
1.1.1 什么是分布式系统	2
1.1.2 集中式系统与分布式系统	2
1.1.3 如何设计分布式系统	4
1.1.4 分布式系统所面临的挑战	4
1.2 线程	5
1.2.1 什么是线程	5
1.2.2 进程和线程	6
1.2.3 线程和纤程	7
1.2.4 编程语言中的线程对象	7
1.2.5 SimpleThreads 示例	11
1.3 通信	13
1.3.1 网络 I/O 模型的演进	13
1.3.2 远程过程调用（RPC）	28
1.3.3 面向消息的通信	35
1.4 一致性	38
1.4.1 以数据为中心的一致性模型	38
1.4.2 以客户为中心的一致性	39
1.5 容错性	40
1.5.1 基本概念	41
1.5.2 故障分类	41
1.5.3 使用冗余来掩盖故障	42
1.5.4 分布式提交	42





1.6	CAP 理论.....	46
1.6.1	什么是 CAP 理论.....	47
1.6.2	为什么 CAP 只能三选二.....	48
1.6.3	CAP 常见模型.....	49
1.6.4	CAP 的意义.....	50
1.6.5	CAP 最新发展.....	50
1.7	安全性.....	51
1.7.1	基本概念.....	52
1.7.2	加密算法.....	54
1.7.3	安全通道.....	57
1.7.4	访问控制.....	66
1.8	并发.....	68
1.8.1	线程与并发.....	69
1.8.2	并发与并行.....	69
1.8.3	并发带来的风险.....	70
1.8.4	同步（Synchronization）.....	72
1.8.5	原子访问（Atomic Access）.....	77
1.8.6	无锁化设计提升并发能力.....	78
1.8.7	缓存提升并发能力.....	78
1.8.8	更细颗粒度的并发单元.....	79
第 2 章	分布式系统架构体系.....	80
2.1	基于对象的体系结构.....	81
2.1.1	分布式对象.....	81
2.1.2	Java RMI.....	82
2.2	面向服务的架构（SOA）.....	85
2.2.1	SOA 的基本概念.....	86
2.2.2	基于 Web Services 的 SOA.....	88
2.2.3	SOA 的演变.....	103
2.3	REST 风格的架构.....	103
2.3.1	什么是 REST.....	103
2.3.2	REST 有哪些特征.....	104





2.3.3	Java 实现 REST 的例子	106
2.3.4	REST API 最佳实践	116
2.4	微服务架构 (MSA)	119
2.4.1	什么是 MSA	119
2.4.2	MSA 与 SOA	121
2.4.3	何时采用 MSA	124
2.4.4	如何构建微服务	125
2.5	容器技术	129
2.5.1	虚拟化技术	129
2.5.2	容器与虚拟机	130
2.5.3	基于容器的持续部署	132
2.6	Serverless 架构	140
2.6.1	什么是 Serverless 架构	141
2.6.2	Serverless 典型的应用场景	142
2.6.3	Serverless 架构原则	144
2.6.4	例子：使用 Serverless 实现游戏全球同服	146
第 3 章	分布式消息服务	152
3.1	分布式消息概述	153
3.1.1	基本概念	153
3.1.2	使用场景	153
3.1.3	常用技术	154
3.2	Apache ActiveMQ	154
3.2.1	例子：producer-consumer	154
3.2.2	例子：使用 JMX 来监控 ActiveMQ	155
3.2.3	例子：使用 Java 实现 producer-consumer	157
3.3	RabbitMQ	162
3.3.1	例子：Work Queues	162
3.3.2	例子：Publish/Subscribe	168
3.3.3	例子：Routing	172
3.3.4	例子：Topics	176
3.3.5	例子：RPC	181



3.4	Apache RocketMQ.....	186
3.4.1	例子：使用 Java 实现 producer-consumer.....	189
3.4.2	RocketMQ 最佳实践.....	193
3.5	Apache Kafka	198
3.5.1	Apache Kafka 的核心概念	199
3.5.2	Apache Kafka 的使用场景	202
3.6	实战：基于 JMS 的消息发送和接收	203
3.6.1	项目概述	203
3.6.2	项目配置	205
3.6.3	编码实现	209
3.6.4	运行	215
第 4 章	分布式计算	221
4.1	分布式计算概述	222
4.1.1	使用场景	222
4.1.2	常用技术	222
4.2	MapReduce	223
4.2.1	MapReduce 简介	223
4.2.2	MapReduce 的编程模型	223
4.2.3	MapReduce 接口实现	228
4.2.4	MapReduce 的使用技巧	234
4.3	Apache Hadoop.....	236
4.3.1	Apache Hadoop 的核心组件.....	237
4.3.2	例子：词频统计 WordCount 程序	238
4.4	Spark.....	240
4.4.1	Spark 简介	240
4.4.2	Spark 与 Hadoop 的关系	241
4.4.3	Spark 2.0 的新特性	242
4.4.4	Spark 集群模式	246
4.5	Mesos.....	248
4.5.1	Mesos 简介	249
4.5.2	设计高可用的 Mesos framework.....	250

4.6 实战：基于 Spark 的词频统计	257
4.6.1 项目概述	257
4.6.2 项目配置	257
4.6.3 编码实现	258
4.6.4 运行	259
 第 5 章 分布式存储	 262
5.1 分布式存储概述	263
5.1.1 使用场景	263
5.1.2 常用技术	263
5.2 Bigtable	264
5.2.1 Bigtable 的数据模型	264
5.2.2 Bigtable 的实现	266
5.2.3 Bigtable 的性能优化	270
5.3 Apache HBase	273
5.3.1 Apache HBase 的基本概念	274
5.3.2 Apache HBase 的架构	281
5.4 Apache Cassandra	296
5.4.1 Apache Cassandra 简介	296
5.4.2 Apache Cassandra 的应用场景	299
5.4.3 Apache Cassandra 的架构和数据模型	300
5.4.4 用于配置 Apache Cassandra 的核心组件	301
5.5 Memcached	302
5.5.1 Memcached 简介	303
5.5.2 Memcached 的架构	303
5.5.3 Memcached 客户端	305
5.6 Redis	313
5.6.1 Redis 简介	313
5.6.2 Redis 的下载与简单使用	314
5.6.3 Redis 的数据类型及抽象	314
5.7 MongoDB	334
5.7.1 MongoDB 简介	334

5.7.2	MongoDB 核心概念	335
5.7.3	MongoDB 的数据模型	340
5.7.4	示例：Java 连接 MongoDB	354
5.8	实战：基于 Redis 的分布式锁	355
5.8.1	项目概述	355
5.8.2	项目配置	356
5.8.3	编码实现	357
5.8.4	运行	360
第 6 章	分布式监控	364
6.1	分布式监控概述	365
6.1.1	使用场景	365
6.1.2	常用技术	365
6.2	Nagios	365
6.2.1	Nagios 监控	366
6.2.2	Nagios 插件	384
6.3	Zabbix	386
6.3.1	Zabbix 对容器的支持	386
6.3.2	Zabbix 的基本概念	389
6.4	Consul	399
6.4.1	Consul 架构	400
6.4.2	Consul agent	401
6.5	ZooKeeper	411
6.5.1	ZooKeeper 简介	411
6.5.2	ZooKeeper 内部工作原理	415
6.5.3	例子：ZooKeeper 实现 barrier 和 producer-consumer queue	419
6.6	实战：基于 ZooKeeper 的服务注册和发现	426
6.6.1	项目概述	426
6.6.2	项目配置	427
6.6.3	编码实现	428
6.6.4	运行	433

第 7 章 分布式版本控制系统.....	435
7.1 分布式版本控制系统概述.....	436
7.1.1 集中式与分布式	436
7.1.2 分布式版本控制系统的核心概念.....	437
7.2 Bazaar	437
7.2.1 Bazaar 的核心概念	437
7.2.2 Bazaar 的使用	438
7.3 Mercurial.....	443
7.3.1 Mercurial 的核心概念.....	444
7.3.2 Mercurial 的使用.....	447
7.4 Git	454
7.4.1 Git 的基础概念	454
7.4.2 Git 的使用	457
7.5 Git Flow——团队协作最佳实践	483
7.5.1 分支定义	483
7.5.2 新功能开发工作流	484
7.5.3 Bug 修复工作流.....	485
7.5.4 版本发布工作流	485
第 8 章 RESTful API、微服务及容器技术	487
8.1 Jersey	488
8.1.1 Jersey 简介	488
8.1.2 Jersey 的模块和依赖	488
8.1.3 JAX-RS 核心概念.....	492
8.1.4 例子：用 SSE 构建实时 Web 应用	503
8.2 Spring Boot.....	511
8.2.1 Spring Boot 简介	512
8.2.2 Spring Boot 的安装.....	513
8.2.3 Spring Boot 的使用	518
8.2.4 Spring Boot 的属性与配置	524
8.3 Docker.....	529
8.3.1 Docker 简介.....	529

8.3.2	Docker 的核心组成、架构及工作原理	529
8.3.3	Docker 的使用	535
8.4	实战：基于 Docker 构建、运行、发布微服务	537
8.4.1	编写微服务	537
8.4.2	微服务容器化	538
8.4.3	构建 Docker image	538
8.4.4	运行 image	540
8.4.5	访问应用	541
8.4.6	发布微服务	541



第 1 章

分布式系统基础知识

1.1 概述

毫无疑问，计算机改变了人类的工作和生活方式，而计算机系统也正在进行一场变革。无论是手机应用，还是智能终端，都离不开背后那个神秘的巨人——分布式系统。正是那些看不见的分布式系统，每天处理着数以亿计的计算，提供可靠而稳定的服务。

本章就揭开分布式系统的神秘面纱。

1.1.1 什么是分布式系统

《分布式系统原理与范型》一书中是这样定义分布式系统的：

“分布式系统是若干独立计算机的集合，这些计算机对于用户来说就像单个系统。”

这里面包含了两个要点：

- 硬件独立；
- 软件统一。

什么是硬件独立？所谓硬件独立，是指机器本身是独立的。一个大型的分布式系统，由若干计算机组成系统的基础设施。而软件统一，是指对于用户来说，用户就像跟单个系统打交道。就好比我们每天上网看视频，视频网站对我们来说就是一个系统软件，它们背后是如何运作的、部署了几台服务器、每台服务器是干什么的，这些对用户来说是不可见的。用户不关心背后的这些服务器，用户所关心的是，今天能看什么样的节目、视频运行是否流畅、清晰度如何等。

软件统一的另外一个方面是指，分布式系统的扩展和升级都比较容易。分布式系统中的某些节点发生故障，不会影响整体系统的可用性。用户和应用程序交互时，不会察觉哪些部分正在替换或维修，也不会感知新加入的部分。

1.1.2 集中式系统与分布式系统

集中式系统主要部署在 HP、IBM、Sun 等小型机以上档次的服务器中，把所有的功能都集成到主服务器上（这对服务器的要求很高）。它们的主要特色在于年宕机时间只有几小时，所以又统称为 z 系列（zero，零）。AS/400 主要应用在银行和制造业，还用于 Domino，主要的技术包括 TIMI（技术独立机器界面）和单级存储。有了 TIMI，可以实现硬件与软件相互独立。RS/6000 比较常见，用于科学计算和事务处理等。这类主机的单机性能一般都很强，带多个终端。终端没有数据处理能力，运算全部在主机上进行。现在的银行系统大部分都是集中式系统。

此外，这种系统在大型企业、科研单位、军队和政府等也有应用。

集中式系统主要流行于 20 世纪。现在还在使用集中式系统的很大一部分原因是沿用原来的软件，而这些软件往往很昂贵。集中式系统的优点是便于维护、操作简单。但这样的系统也有缺陷，不出问题还好，一出问题，就会造成单点故障，所有功能就都不能正常工作了，所谓“一荣俱荣，一损俱损”。由于集中式系统的相关技术只被少数厂商所掌握，个人要对这些系统进行扩展和升级往往比较麻烦，一般的企业级应用很少会用到集中式系统。图 1-1 是一个典型的集中式系统的示意图。

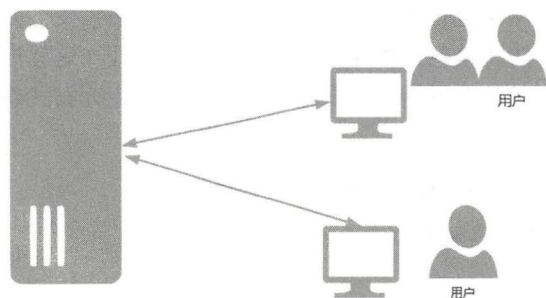


图 1-1 集中式系统示意图

分布式系统恰恰相反。分布式系统通过中间件对现有计算机的硬件能力和相应的软件功能进行重新配置和整合。它是一种多处理器的计算机系统，各处理器通过网络构成统一的系统。系统采用分布式计算结构，即把原来系统内中央处理器处理的任务分散交给相应的处理器，实现不同功能的多个处理器，这些处理器的相互协调，共享系统的外设与软件。这样就加快了系统的处理速度，简化了主机的逻辑结构。它甚至不需要很高的配置，一些“退休”的低配置机器也能被重新纳入分布式系统中，这样无疑降低了硬件成本且易于维护。同时，分布式系统往往由多台主机组成，任何一台主机宕机都不影响整个系统的使用，所以分布式系统的可用性比较高。图 1-2 是一个典型的分布式系统的示意图。



图 1-2 分布式系统示意图

正由于分布式系统的这些优点，使得分布式系统的应用越来越广泛。可以说，这也代表了未来应用的发展趋势。

1.1.3 如何设计分布式系统

设计分布式系统的本质就是“合理地将一个系统拆分成多个子系统并部署到不同的机器上”，所以首先要考虑的问题是如何合理地将系统进行拆分。由于拆分后的各个子系统不可能孤立地存在，必然要通过网络进行连接交互，它们之间如何通信变得尤为重要。当然，在通信过程中要识别“敌我”，防止信息在传递过程中被拦截和篡改（这就涉及安全问题了）。分布式系统如果要适应不断增长的业务需求，就需要考虑扩展性。分布式系统还必须保证可靠性和数据的一致性。

概括起来，在设计分布式系统时，应考虑以下问题：

- 如何将系统拆分为子系统？
- 如何规划子系统间的通信？
- 如何考虑通信过程中的安全？
- 如何让子系统可以扩展？
- 如何保证子系统的可靠性？
- 如何实现数据的一致性？

本书的大部分内容都将针对分布式系统中常见的问题进行探讨。

1.1.4 分布式系统所面临的挑战

分布式系统是难以理解、设计、构建和管理的，它们将比单个机器多数倍的变量引入设计中，使得应用程序的问题根源更难被发现。SLA（Service-Level Agreement，服务水平协议）是衡量停机和/或性能下降的标准。大多数现代应用程序都有一个期望的弹性 SLA，通常按“9”的数量增加（如每月 99.9%或 99.99%可用性）。

设计分布式系统的挑战如下。

- **异构性**：由于分布式系统基于不同的网络、操作系统、计算机硬件和编程语言构造，必须有一种通用的网络通信协议来屏蔽异构系统之间的差异。一般由中间件来处理这些差异。
- **缺乏全球时钟**：在程序需要协作时，通过交换消息来协调它们的动作。紧密的协调经常依赖于对程序动作发生时间的共识。但是，实际网络上计算机同步时钟的准确性受到了

极大的限制，即没有一个正确时间的全局概念。这是通过网络发送消息作为唯一的通信方式这一事实带来的直接结果。

- **一致性**：数据被分散或复制到不同的机器上，如何保证各台主机之间的数据的一致性是一个难点。
- **故障的独立性**：任何计算机都有可能发生故障，且故障不尽相同。故障出现的时间也是相互独立的。一般分布式系统允许出现部分故障而不影响整个系统的正常使用。
- **并发**：使用分布式系统的目的是更好地共享资源，所以系统中的每个资源都必须被设计成在并发环境中是安全的。
- **透明性**：分布式系统中任何组件的故障，或者主机的升级、迁移，对于用户来说都是不可见的。
- **开放性**：分布式系统由不同的程序员来编写不同的组件，组件最终要集成为一个系统，所以组件所发布的接口必须遵守一定的规范且能够被互相理解。
- **安全性**：加密用于给共享资源提供适当的保护，在网络上传递的所有敏感信息都需要进行加密。拒绝服务攻击仍然是一个有待解决的问题。
- **可扩展性**：系统要设计成随着业务量的增加而相应地扩展，以提供对应的服务。

1.2 线程

在早期的计算机操作系统中，拥有资源且独立运行的基本单位是进程。随着计算机技术的发展，进程出现了很多弊端。例如，由于进程是资源拥有者，创建、撤销与切换存在较大的时空开销，需要引入轻量级进程；由于对称多处理机（Symmetric Multi-Processor, SMP）出现，可以满足多个运行单位，而多个进程并行开销过大。

因此，在 20 世纪 80 年代，出现了能独立运行的基本单位——线程（Thread）。

1.2.1 什么是线程

线程是程序执行流的最小单元。一个标准的线程由线程 ID、当前指令指针（PC）、寄存器集合和堆栈组成。另外，线程是进程中的一个实体，是被系统独立调度和分派的基本单位。线程只拥有少量在运行中必不可少的资源，但它可与同属一个进程的其他线程共享进程所拥有的全部资源。一个线程可以创建和撤销另一个线程，同一进程中的多个线程之间可以并发执行。由于线程之间相互制约，线程在运行中呈现出间断性。

线程有三种基本状态：就绪、阻塞和运行。

线程的状态图如图 1-3 所示。

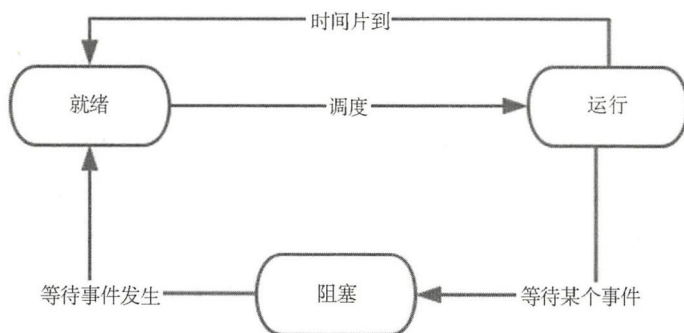


图 1-3 线程的状态图

就绪状态是指线程具备运行的所有条件，逻辑上可以运行，在等待处理机；运行状态是指线程占有处理机，正在运行；阻塞状态是指线程在等待一个事件（如某个信号量），逻辑上不可执行。每一个程序至少有一个线程，若程序只有一个线程，那就是程序本身。

线程是程序中一个单一的顺序控制流程，是进程内一个相对独立的、可调度的执行单元。在单个程序中同时运行多个线程完成不同的工作，称为多线程。在大多数情况下，多线程能提升程序的性能。

1.2.2 进程和线程

进程和线程是并发编程的两个基本的执行单元。在大多数编程语言中，并发编程主要涉及线程。

一个计算机系统通常有许多活动的进程和线程。在给定的时间内，每个处理器中只能有一个线程得到真正的运行。对于单核处理器来说，处理时间是通过时间切片在进程和线程之间进行共享的。

进程有一个独立的执行环境。进程通常有一个完整的、私人的基本运行时资源。每个进程都有自己的内存空间。操作系统的进程表（Process Table）存储了 CPU 寄存器值、内存映像、打开的文件、统计信息和特权信息等。进程一般定义为执行中的程序，也就是当前操作系统的某个虚拟处理器上运行的一个程序。多个进程并发共享同一个 CPU 和其他硬件资源，操作系统支持进程之间的隔离。这种并发透明性需要付出相对较高的代价。

进程往往被等同为程序或应用程序。然而，用户看到的一个单独的应用程序可能实际上是一组合作的进程。大多数操作系统都支持进程间通信（Inter Process Communication, IPC），如管道和 socket。IPC 不仅用于同一个系统的进程之间的通信，也用于不同系统的进程之间的通信。

线程，有时被称为轻量级进程（Lightweight Process, LWP）。进程和线程都提供了一个执行环境，但创建一个新的线程比创建一个新的进程需要更少的资源。线程系统一般只维护用来让多个线程共享 CPU 所必需的最少量信息，特别是线程上下文（Thread Context）中一般只包含 CPU 上下文及某些其他线程管理信息。通常忽略那些对于多线程管理而言不必要的信息。这样，单个进程中防止数据遭到某些线程不合法的访问的任务就完全落在了应用程序开发人员的肩上。线程不像进程那样彼此隔离，以及受到操作系统的自动保护，所以在多线程程序开发过程中需要开发人员做更多的努力。

线程存在于进程中，每个进程至少有一个线程。线程共享进程的资源，包括内存和打开的文件，尽管这使得工作变得高效，但也存在一个潜在的问题——通信。关于通信的内容，会在后面的章节中讲述。

现在多核处理器或多进程的计算机系统越来越流行，这大大增强了系统的进程和线程的并发执行能力。即便在没有多处理器或多进程的系统中，并发仍然是可能的。关于并发的内容，会在后面章节中讲述。

1.2.3 线程和纤程

为了提高并发量，某些编程语言中提供了“纤程”（Fiber）的概念，比如 Golang 的 goroutine、Erlang 风格的 actor。Java 语言虽然没有定义纤程，但仍有一些第三方库供选择，比如 Quasar。纤程可以理解为比线程颗粒度更细的并发单元。

纤程是以用户方式代码来实现的，并不受操作系统内核管理，所以内核并不知道纤程，也就无法对纤程实现调度。纤程是根据用户定义的算法来调度的。因此，就内核而言，纤程采用了非抢占式调度方式，而线程是抢占式调度的。

一个线程可以包含一个或多个纤程。线程每次执行哪一个纤程的代码，是由用户来决定的。

所以，对于开发人员来说，使用纤程可以获得更大的并发量，但同时要面临自己实现调度纤程的复杂度。

1.2.4 编程语言中的线程对象

在面向对象语言开发中，每个线程都与 Thread 类的一个实例相关联。由于 Java 语言较流行，下面将用 Java 来实现并使用线程对象，作为并发应用程序的基本原型。

1. 定义和启动一个线程

Java 中有两种创建 Thread 实例的方式。

- 提供 Runnable 对象。Runnable 接口定义了一个方法 run，用来包含线程要执行的代码。HelloRunnable 示例如下。

```
public class HelloRunnable implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("Hello from a runnable!");  
    }  
    public static void main(String[] args) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```

- 继承 Thread 类。Thread 类本身是 Runnable 的实现，只是它的 run 方法什么都没干。HelloThread 示例如下。

```
public class HelloThread extends Thread {  
    @Override  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
    public static void main(String[] args) {  
        (new HelloThread()).start();  
    }  
}
```

请注意，这两个例子调用 start 来启动线程。

第一种方式使用 Runnable 对象，在实际应用中更普遍，因为 Runnable 对象可以继承 Thread 以外的类。第二种方式在简单的应用程序中更容易使用，但受理的任务类必须是一个 Thread 类的后代。本书推荐使用第一种方式，将 Runnable 任务从 Thread 对象中分离来执行任务。这样不仅更灵活，而且适用于高级线程管理 API。

Thread 类还定义了大量的方法用于线程管理。

2. 使用 sleep 来暂停执行

Thread.sleep 可以让当前线程执行暂停一个时间段，这样处理器的时间就可以给其他线程使用。

`sleep` 有两种重载形式：一种是指定睡眠时间为毫秒级，另一种是指定睡眠时间为纳秒级。然而，这些睡眠时间不能保证是精确的，因为它们是由操作系统提供的并受其限制。此外，睡眠周期也可以通过中断来终止，我们将在后面的章节中看到。

SleepMessages 示例——使用 `sleep` 每隔 4 秒打印一次消息：

```
public class SleepMessages {

    public static void main(String[] args) throws InterruptedException
    {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too" };
        for (int i = 0; i < importantInfo.length; i++) {
            // 暂停 4 秒
            Thread.sleep(4000);
            // 打印消息
            System.out.println(importantInfo[i]);
        }
    }
}
```

请注意，`main` 声明抛出 `InterruptedException`。如果 `sleep` 是激活的，若有另一个线程中断当前线程，则 `sleep` 抛出异常。因为该应用程序还没有定义另一个线程来引起中断，所以考虑捕捉 `InterruptedException`。

3. 中断 (interrupt)

中断表明一个线程应该停止它正在做和将要做的事。线程通过 `Thread` 对象调用 `interrupt` 实现中断。为了使中断机制正常工作，被中断的线程必须支持自己的中断。

支持中断

如何实现线程支持自己的中断？这要看它目前正在做什么。如果线程调用方法频繁抛出 `InterruptedException` 异常，那么它只要在 `run` 方法捕获异常之后返回即可。例如：

```
for (int i = 0; i < importantInfo.length; i++) {
    // 暂停 4 秒
    try {
        Thread.sleep(4000);
```

```
    } catch (InterruptedException e) {  
        // 已经中断，不需要更多信息  
        return;  
    }  
    // 打印消息  
    System.out.println(importantInfo[i]);  
}
```

很多方法都会抛出 `InterruptedException`，如 `sleep` 被设计成在收到中断时立即取消它们当前的操作并返回。

若线程长时间没有调用方法抛出 `InterruptedException`，那么它必须定期调用 `Thread.interrupted`，该方法在收到中断后返回 `true`。

```
for (int i = 0; i < inputs.length; i++) {  
    heavyCrunch(inputs[i]);  
    if (Thread.interrupted()) {  
        // 已经中断，不需要更多信息  
        return;  
    }  
}
```

在这个简单的例子中，代码简单地测试该中断，如果已接收中断线程就退出。在更复杂的应用程序中，它可能会更有意义地抛出一个 `InterruptedException`：

```
if (Thread.interrupted()) {  
    throw new InterruptedException();  
}
```

中断状态标志

中断机制是使用被称为中断状态的内部标志实现的。调用 `Thread.interrupt` 可以设置该标志。当一个线程通过调用静态方法 `Thread.interrupted` 来检查中断时，中断状态被清除。非静态 `isInterrupted` 方法用于线程查询另一个线程的中断状态，而不会改变中断状态标志。

按照惯例，任何方法因抛出一个 `InterruptedException` 而退出都会清除中断状态。当然，它可能因为另一个线程调用 `interrupt` 而让那个中断状态立即被重新设置。

4. join 方法

`join` 方法允许一个线程等待另一个线程完成。假设 `t` 是一个正在执行的 `Thread` 对象，那么

```
t.join();
```


会导致当前线程暂停执行直到 t 线程终止。join 方法允许程序员指定一个等待周期，与 sleep 一样，等待时间依赖于操作系统的时间，同时不能假设 join 方法的等待时间是精确的。

像 sleep 一样，join 方法通过 InterruptedException 退出来响应中断。

1.2.5 SimpleThreads 示例

SimpleThreads 示例有两个线程。第一个线程是每个 Java 应用程序都有的主线程。主线程创建 Runnable 对象的 MessageLoop，并等待它完成。如果 MessageLoop 需要很长时间才能完成，主线程就中断它。

该 MessageLoop 线程打印出一系列消息。如果中断之前就已经打印了所有消息，则 MessageLoop 线程打印一条消息并退出。

```
public class SimpleThreads {

    // 显示当前执行线程的名称、信息
    static void threadMessage(String message) {
        String threadName =
            Thread.currentThread().getName();
        System.out.format("%s: %s%n",
                           threadName,
                           message);
    }

    private static class MessageLoop
        implements Runnable {
        public void run() {
            String importantInfo[] = {
                "Mares eat oats",
                "Does eat oats",
                "Little lambs eat ivy",
                "A kid will eat ivy too"
            };
            try {
                for (int i = 0; i < importantInfo.length; i++) {
                    // 暂停 4 秒
                    Thread.sleep(4000);
                    // 打印消息
```

```
        threadMessage(importantInfo[i]);
    }
} catch (InterruptedException e) {
    threadMessage("I wasn't done!");
}
}
}

public static void main(String args[])
    throws InterruptedException {

    // 在中断 MessageLoop 线程（默认为 1 小时）前延迟一段时间（单位是毫秒）
    long patience = 1000 * 60 * 60;
    // 如果命令行参数出现
    // 设置 patience 的时间值
    // 单位是秒
    if (args.length > 0) {
        try {
            patience = Long.parseLong(args[0]) * 1000;
        } catch (NumberFormatException e) {
            System.err.println("Argument must be an integer.");
            System.exit(1);
        }
    }

    threadMessage("Starting MessageLoop thread");
    long startTime = System.currentTimeMillis();
    Thread t = new Thread(new MessageLoop());
    t.start();

    threadMessage("Waiting for MessageLoop thread to finish");

    // 循环，直到 MessageLoop 线程退出
    while (t.isAlive()) {
        threadMessage("Still waiting...");

        // 最长等待 1 秒
        // 交给 MessageLoop 线程来完成
        t.join(1000);
    }
}
```



```
        if (((System.currentTimeMillis() - startTime) > patience)
            && t.isAlive()) {
            threadMessage("Tired of waiting!");
            t.interrupt();

            // 等待
            t.join();
        }
    }
    threadMessage("Finally!");
}
```

对 Java 的 Thread 类感兴趣的读者可以查看其在线 API: <https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html>。

上面例子的代码可以在 <https://github.com/waylau/distributed-systems-technologies-and-cases-analysis> 的 distributed-systems-java-demos 程序的 com.waylau.essentialjava.thread 包下找到。

1.3 通信

进程间的通信是一切分布式系统的核心。如果没有通信机制，分布式系统的各个子系统将是“一盘散沙”，毫无作用。本节介绍常用的通信方式。

1.3.1 网络 I/O 模型的演进

1. 同步和异步

同步和异步描述的是用户线程与内核的交互方式：

- **同步**是指用户线程发起 I/O 请求后需要等待，或者轮询内核 I/O 操作完成后才能继续执行；
- **异步**是指用户线程发起 I/O 请求后仍继续执行，当内核 I/O 操作完成后会通知用户线程，或者调用用户线程注册的回调函数。

2. 阻塞和非阻塞

阻塞和非阻塞描述的是用户线程调用内核 I/O 操作的方式：

- **阻塞**是指 I/O 操作需要彻底完成后才返回用户空间；

- **非阻塞**是指 I/O 操作被调用后立即返回给用户一个状态值，无须等到 I/O 操作彻底完成。

一个 I/O 操作其实分成了两个步骤：发起 I/O 请求和实际的 I/O 操作。

阻塞 I/O 和非阻塞 I/O 的区别在于第一步，也就是发起 I/O 请求是否会被阻塞。如果阻塞直到完成，就是传统的阻塞 I/O，如果不阻塞，就是非阻塞 I/O。

同步 I/O 和异步 I/O 的区别在于第二个步骤是否阻塞，如果实际的 I/O 读写阻塞请求进程，就是同步 I/O。

3. UNIX I/O 模型

UNIX 下共有 5 种 I/O 模型：

- 阻塞 I/O；
- 非阻塞 I/O；
- I/O 复用（select 和 poll）；
- 信号驱动 I/O（SIGIO）；
- 异步 I/O（Posix.1 的 aio_系列函数）。

注：读者若想深入了解 UNIX 的网络知识，推荐阅读 W.Richard Stevens 的 *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI*。本节只简单介绍了这 5 种模型，文中的图例也引用自该书。

阻塞 I/O 模型

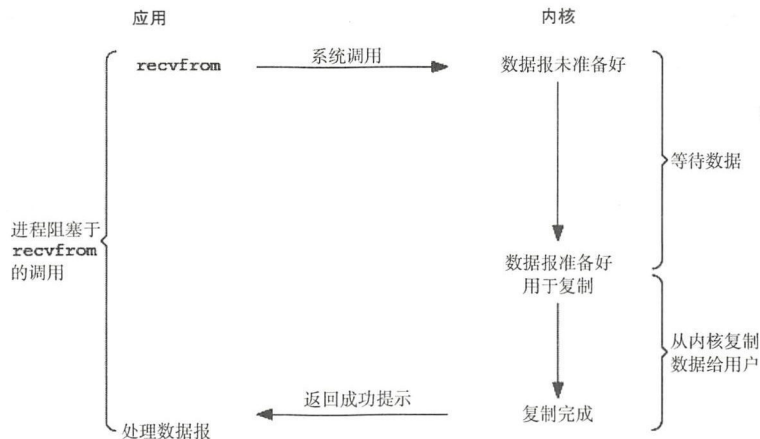
请求无法立即完成则保持阻塞。

- **阶段 1：**等待数据就绪。网络 I/O 的情况就是等待远端数据陆续抵达；磁盘 I/O 的情况就是等待磁盘数据从磁盘上读取到内核态内存中。
- **阶段 2：**数据复制。出于系统安全考虑，用户态的程序没有权限直接读取内核态内存，因此内核负责把内核态内存中的数据复制一份到用户态内存中。

阻塞 I/O 模型如图 1-4 所示。

本节中将 `recvfrom` 函数视为系统调用。一般 `recvfrom` 函数的实现都有一个从应用程序进程中运行到内核中运行的切换，一段时间后再跟一个返回应用进程的切换。

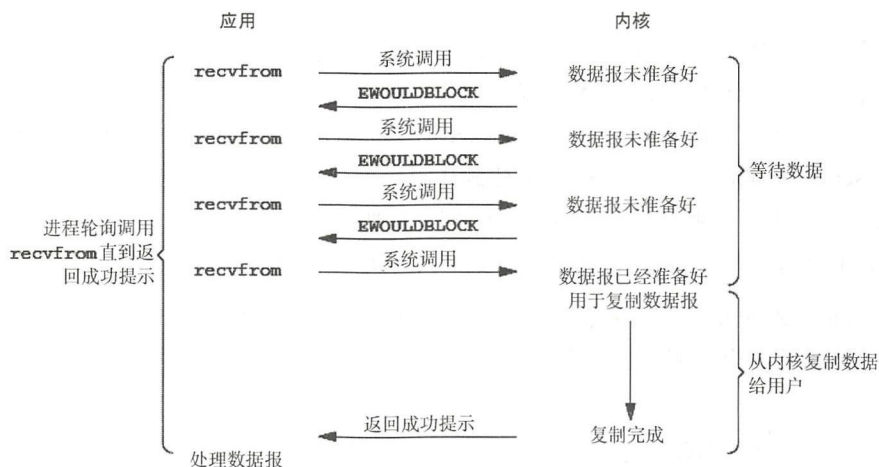
在图 1-4 中，进程阻塞的整段时间是指从调用 `recvfrom` 函数开始到它返回的这段时间，当进程返回成功提示时，应用进程开始处理数据报。



非阻塞 I/O 模型

- socket 设置为 NONBLOCK（非阻塞）就是告诉内核，当所请求的 I/O 操作无法完成时，不要让进程进入睡眠状态，而是立刻返回一个错误码（EWOULDBLOCK），这样请求就不会阻塞；
- I/O 操作函数将不断地测试数据是否已经准备好，如果没有准备好，则继续测试，直到数据准备好为止。在整个 I/O 请求的过程中，虽然用户线程每次发起 I/O 请求后可以立即返回，但是为了等到数据，仍需轮询、重复请求，而这是对 CPU 时间的极大浪费。
- 数据准备好了，从内核复制到用户空间。

非阻塞 I/O 模型如图 1-5 所示。



一般很少直接使用这种模型，而是在其他 I/O 模型中使用非阻塞 I/O 这一特性。这种方式对单个 I/O 请求的意义不大，但给 I/O 复用铺平了道路。

I/O 复用模型

I/O 复用会用到 `select` 或 `poll` 函数，在这两个系统调用中的某一个上阻塞，而不是阻塞于真正的 I/O 系统调用。函数也会使进程阻塞，但和阻塞 I/O 不同的是，这两个函数可以同时阻塞多个 I/O 操作。而且可以同时对多个读操作、多个写操作的 I/O 函数进行检测，直到有数据可读或可写时，才真正调用 I/O 操作函数。

I/O 复用模型如图 1-6 所示。

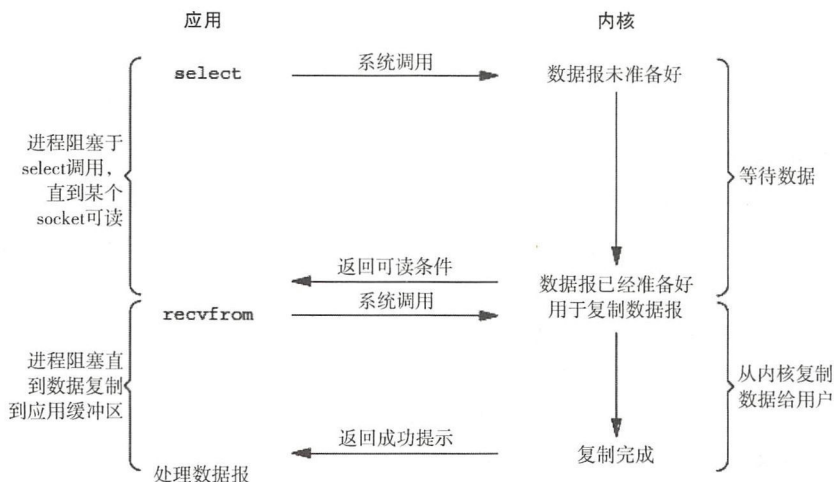


图 1-6 I/O 复用模型

从流程上看，使用 `select` 函数进行 I/O 请求和同步阻塞模型没有太大的区别，甚至还多了添加监视 `socket`，以及调用 `select` 函数的额外操作，效率更差。但是，使用 `select` 函数最大的优势是用户可以在一个线程内同时处理多个 `socket` 的 I/O 请求。用户可以注册多个 `socket`，然后不断地调用 `select` 来读取被激活的 `socket`，达到在同一个线程内同时处理多个 I/O 请求的目的。而在同步阻塞模型中，必须通过多线程的方式才能达到这个目的。

I/O 复用模型使用 Reactor 设计模式实现了这一机制。

调用 `select` 或 `poll` 函数的方法由一个用户态线程负责轮询多个 `socket`，直到阶段 1 的数据就绪，再通知实际的用户线程执行阶段 2 的复制操作。通过一个专职的用户态线程执行非阻塞 I/O 轮询，模拟实现阶段 1 的异步化。

信号驱动 I/O (SIGIO) 模型

首先，我们允许 `socket` 进行信号驱动 I/O，并通过调用 `sigaction` 来安装一个信号处理函数，



进程继续运行并不阻塞。当数据准备好后，进程会收到一个 SIGIO 信号，可以在信号处理函数中调用 `recvfrom` 来读取数据报，并通知主循环数据已准备好被处理，也可以通知主循环，让它来读取数据报。

信号驱动 I/O (SIGIO) 模型如图 1-7 所示。

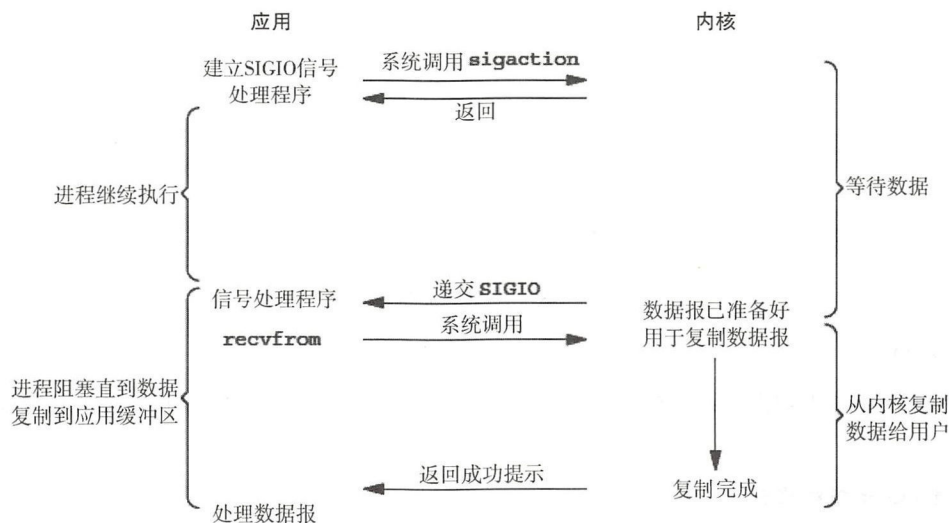


图 1-7 信号驱动 I/O (SIGIO) 模型

异步 I/O 模型

异步 I/O 是 POSIX 规范定义的。通常，这些函数会通知内核来启动操作并在整个操作（包括从内核复制数据到我们的缓存中）完成时通知我们。

该模型与信号驱动 I/O (SIGIO) 模型的不同点在于，信号驱动 I/O (SIGIO) 模型告诉我们 I/O 操作何时可以启动，而异步 I/O 模型告诉我们 I/O 操作何时完成。

调用 `aio_read` 函数，告诉内核传递描述字、缓存区指针、缓存区大小和文件偏移，然后立即返回，我们的进程不阻塞于等待 I/O 操作的完成。当内核将数据复制到缓存区后，才会生成一个信号来通知应用程序。

异步 I/O 模型如图 1-8 所示。

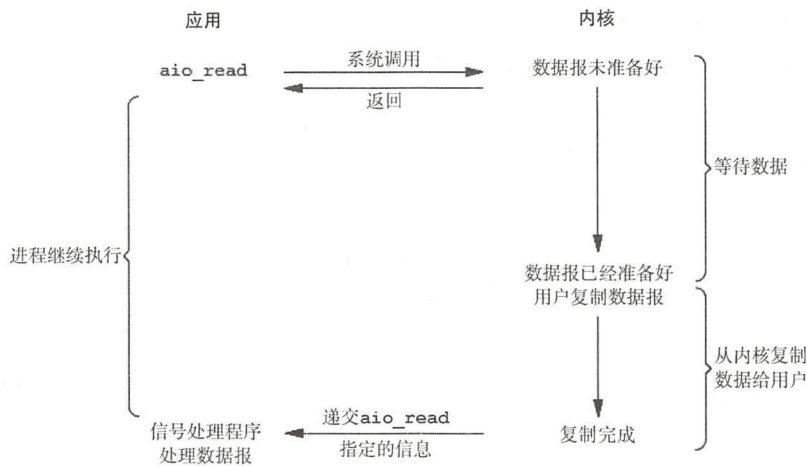


图 1-8 异步 I/O 模型

异步 I/O 模型使用 Proactor 设计模式实现了这一机制。¹

异步 I/O 模型告知内核：当整个过程（包括阶段 1 和阶段 2）全部完成时，通知应用程序来读数据。

几种 I/O 模型的比较

前 4 种模型的区别是阶段 1 不相同，阶段 2 基本相同，都是将数据从内核复制到调用者的缓存区。而异步 I/O 的两个阶段都不同于前 4 个模型。几种 I/O 模型的比较如图 1-9 所示。

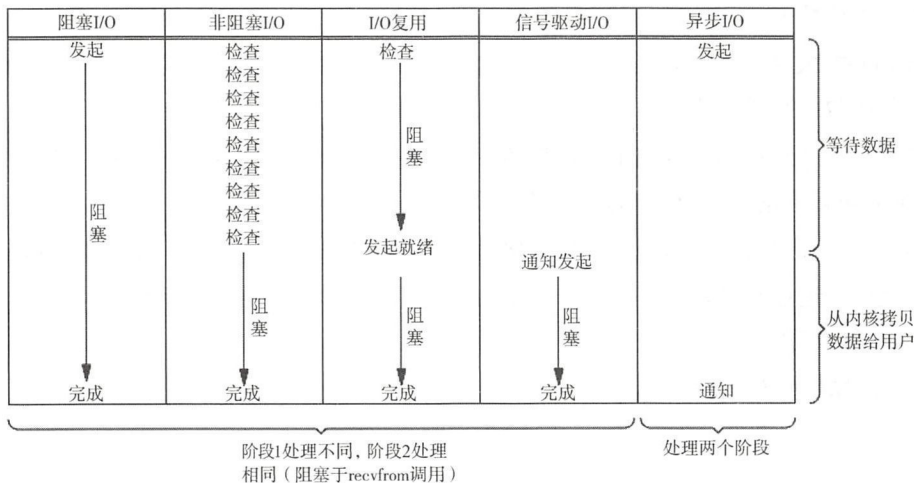


图 1-9 几种 I/O 模型的比较

¹ 有关“Proactor 设计模式”可以参阅 https://en.wikipedia.org/wiki/Proactor_pattern。



同步 I/O 操作引起请求进程阻塞，直到 I/O 操作完成。异步 I/O 操作不引起请求进程阻塞。上面前 4 个模型——阻塞 I/O 模型、非阻塞 I/O 模型、I/O 复用模型和信号驱动 I/O 模型都是同步 I/O 模型，而异步 I/O 模型才是真正的异步 I/O。

4. 常见 Java I/O 模型

在了解了 UNIX 的 I/O 模型之后，就能明白其实 Java 的 I/O 模型也是类似的。

“阻塞 I/O”模型

下面的 EchoServer 是一个简单的阻塞 I/O 例子，服务器启动后，等待客户端连接。在客户端连接服务器后，服务器就阻塞读写数据流。

EchoServer 代码：

```
public class EchoServer {
    public static int DEFAULT_PORT = 7;

    public static void main(String[] args) throws IOException {

        int port;

        try {
            port = Integer.parseInt(args[0]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }

        try {
            ServerSocket serverSocket =
                new ServerSocket(port);
            Socket clientSocket = serverSocket.accept();
            PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
        } {
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                out.println(inputLine);
            }
        }
    }
}
```



```
        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen
                                on port "+ port + " or listening for a connection");
            System.out.println(e.getMessage());
        }
    }
}
```

改进为“阻塞 I/O+多线程”模型

使用多线程来支持多个客户端访问服务器。

主线程 `MultiThreadEchoServer.java`:

```
public class MultiThreadEchoServer {
    public static int DEFAULT_PORT = 7;

    public static void main(String[] args) throws IOException {

        int port;

        try {
            port = Integer.parseInt(args[0]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }
        Socket clientSocket = null;
        try (ServerSocket serverSocket = new ServerSocket(port);) {
            while (true) {
                clientSocket = serverSocket.accept();

                // 多线程
                new Thread(new EchoServerHandler(clientSocket)).start();
            }
        } catch (IOException e) {
            System.out.println(
                "Exception caught when trying to listen
                on port " + port + " or listening for a connection");
            System.out.println(e.getMessage());
        }
    }
}
```




处理器类 EchoServerHandler.java:

```
public class EchoServerHandler implements Runnable {
    private Socket clientSocket;

    public EchoServerHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }

    @Override
    public void run() {
        try (PrintWriter out = new PrintWriter(clientSocket.getOutputStream(),
true);

            BufferedReader in = new BufferedReader(new InputStreamReader
(clientSocket.getInputStream()));) {

            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                out.println(inputLine);
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

存在问题：每次收到新的连接都要新建一个线程，处理完后销毁线程，代价大。当有大量的短连接出现时，性能比较低。

改进为“阻塞 I/O+线程池”模型

针对上面多线程的模型中出现的线程重复创建、销毁带来的开销问题，可以采用线程池来优化。每次收到新连接后从池中取一个空闲线程进行处理，处理完后再放回池中，重用线程避免了频繁地创建和销毁线程带来的开销。

主线程 ThreadPoolEchoServer.java:

```
public class ThreadPoolEchoServer {
    public static int DEFAULT_PORT = 7;

    public static void main(String[] args) throws IOException {
```





```
int port;

try {
    port = Integer.parseInt(args[0]);
} catch (RuntimeException ex) {
    port = DEFAULT_PORT;
}

ExecutorService threadPool = Executors.newFixedThreadPool(5);
Socket clientSocket = null;
try (ServerSocket serverSocket = new ServerSocket(port);) {
    while (true) {
        clientSocket = serverSocket.accept();

        // 线程池
        threadPool.submit(new Thread(new EchoServerHandler(clientSocket)));
    }
} catch (IOException e) {
    System.out.println(
        "Exception caught when trying to listen
        on port " + port + " or listening for a connection");
    System.out.println(e.getMessage());
}
}
```

存在问题：在大量短连接的场景中性能会提升，因为不用每次都创建和销毁线程，而是重用连接池中的线程。但在大量长连接的场景中，因为线程被连接长期占用，不需要频繁地创建和销毁线程，所以没有什么优势。

虽然这种方法适用于小到中等规模的客户端的并发数，但是如果连接数超过 100000，那么性能将很不理想。

改进为“非阻塞 I/O”模型

“阻塞 I/O+线程池”模型虽然比“阻塞 I/O+多线程”模型在性能方面有所提升，但这两种模型存在一个共同的问题：读和写操作都是同步阻塞的，面对高并发（持续大量连接同时请求）的场景，需要消耗大量的线程来维持连接。CPU 在大量的线程之间频繁切换，性能损耗很大。一旦单机的连接数超过 1 万，甚至达到几万，服务器的性能会急剧下降。

而 NIO 的 Selector 却很好地解决了这个问题，用主线程（一个线程或 CPU 个数的线程）保





持所有的连接，管理和读取客户端连接的数据，将读取的数据交给后面的线程池处理，线程池处理完业务逻辑后，将结果交给主线程发送响应给客户端，少量的线程就可以处理大量连接请求。

Java NIO 由以下几个核心部分组成：

- Channel;
- Buffer;
- Selector。

要使用 Selector，得向 Selector 注册 Channel，然后调用它的 select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，比如新连接进来、数据接收等事件。

主线程 NonBlockingEchoServer.java:

```
public class NonBlockingEchoServer {
    public static int DEFAULT_PORT = 7;

    public static void main(String[] args) throws IOException {

        int port;

        try {
            port = Integer.parseInt(args[0]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }
        System.out.println("Listening for connections on port " + port);

        ServerSocketChannel serverChannel;
        Selector selector;
        try {
            serverChannel = ServerSocketChannel.open();
            InetSocketAddress address = new InetSocketAddress(port);
            serverChannel.bind(address);
            serverChannel.configureBlocking(false);
            selector = Selector.open();
            serverChannel.register(selector, SelectionKey.OP_ACCEPT);
        } catch (IOException ex) {
```





```
        ex.printStackTrace();
        return;
    }

    while (true) {
        try {
            selector.select();
        } catch (IOException ex) {
            ex.printStackTrace();
            break;
        }
        Set<SelectionKey> readyKeys = selector.selectedKeys();
        Iterator<SelectionKey> iterator = readyKeys.iterator();
        while (iterator.hasNext()) {
            SelectionKey key = iterator.next();
            iterator.remove();
            try {
                if (key.isAcceptable()) {
                    ServerSocketChannel server = (ServerSocketChannel)
key.channel();

                    SocketChannel client = server.accept();
                    System.out.println("Accepted connection from " + client);
                    client.configureBlocking(false);
                    SelectionKey clientKey = client.register(selector,
                        SelectionKey.OP_WRITE | SelectionKey.OP_READ);
                    ByteBuffer buffer = ByteBuffer.allocate(100);
                    clientKey.attach(buffer);
                }
                if (key.isReadable()) {
                    SocketChannel client = (SocketChannel) key.channel();
                    ByteBuffer output = (ByteBuffer) key.attachment();
                    client.read(output);
                }
                if (key.isWritable()) {
                    SocketChannel client = (SocketChannel) key.channel();
                    ByteBuffer output = (ByteBuffer) key.attachment();
                    output.flip();
                    client.write(output);
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```





```
        output.compact();
    }
} catch (IOException ex) {
    key.cancel();
    try {
        key.channel().close();
    } catch (IOException cex) {
    }
}
}
}
}
```

改进为“异步 I/O”模式

Java SE 7 之后的版本，引入了对异步 I/O (NIO.2) 的支持，为构建高性能的网络应用提供了一个利器。

主线程 AsyncEchoServer.java:

```
public class AsyncEchoServer {

    public static int DEFAULT_PORT = 7;

    public static void main(String[] args) throws IOException {
        int port;

        try {
            port = Integer.parseInt(args[0]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }

        ExecutorService taskExecutor = Executors.newCachedThreadPool(
            Executors.defaultThreadFactory());

        // 创建异步服务器 socket channel 并绑定到默认组
```





```
try (AsynchronousServerSocketChannel asynchronousServerSocketChannel
= AsynchronousServerSocketChannel.open()) {
    if (asynchronousServerSocketChannel.isOpen()) {

        // 设置一些参数
        asynchronousServerSocketChannel.setOption
(StandardSocketOptions.SO_RCVBUF, 4 * 1024);
        asynchronousServerSocketChannel.setOption
(StandardSocketOptions.SO_REUSEADDR, true);

        // 绑定服务器 socket channel 到本地地址
        asynchronousServerSocketChannel.bind(new InetSocketAddress
(port));

        // 显示等待客户端的信息
        System.out.println("Waiting for connections ...");
        while (true) {
            Future<AsynchronousSocketChannel>
asynchronousSocketChannelFuture = asynchronousServerSocketChannel
                .accept();
            try {
                final AsynchronousSocketChannel asynchronousSocketChannel
= asynchronousSocketChannelFuture
                    .get();
                Callable<String> worker = new Callable<String>() {
                    @Override
                    public String call() throws Exception {
                        String host = asynchronousSocketChannel
.getRemoteAddress().toString();
                        System.out.println("Incoming connection from: "
+ host);

                        final ByteBuffer buffer = ByteBuffer
.allocateDirect(1024);

                        // 发送数据
                        while (asynchronousSocketChannel.read
(buffer).get() != -1) {
```



```
        buffer.flip();
        asynchronousSocketChannel.write(buffer).get();
        if (buffer.hasRemaining()) {
            buffer.compact();
        } else {
            buffer.clear();
        }
    }
    asynchronousSocketChannel.close();
    System.out.println(host + " was successfully served!");
    return host;
}

};
taskExecutor.submit(worker);
} catch (InterruptedException | ExecutionException ex) {
    System.err.println(ex);
    System.err.println("\n Server is shutting down ...");

    // 执行器不再接收新线程并完成队列中所有的线程
    taskExecutor.shutdown();

    // 等待所有线程完成
    while (!taskExecutor.isTerminated()) {
    }
    break;
}

}

} else {
    System.out.println("The asynchronous server-socket channel
cannot be opened!");
}

} catch (IOException ex) {
    System.err.println(ex);
}

}

}
```



1.3.2 远程过程调用（RPC）

1. 进程间通信（IPC）

进程间通信（Inter-Process Communication, IPC）指至少两个进程或线程间传送数据或信号的一些技术或方法。进程是计算机系统分配资源的最小单位。每个进程都有自己的一部分独立的系统资源，彼此是隔离的。为了使不同的进程互相访问资源并协调工作，才有了进程间通信。这些进程可以运行在同一计算机上或有网络连接的不同计算机上。进程间通信技术包括消息传递、同步、共享内存和远程过程调用。IPC 是一种标准的 UNIX 通信机制。

2. 过程调用的类型

在讨论客户机/服务器（C/S）模型和过程调用时，主要有三种不同类型的过程调用。

- 本地过程调用（Local Procedure Call, LPC）：指被调用的过程（函数）与调用过程处于同一个进程中。典型的情况是，调用者通过执行某条机器指令把控制传给新过程，被调用过程保存机器寄存器的值，并在栈顶分配存放其本地变量的空间。
- 同主机间的远程过程调用（Remote Procedure Call, RPC）：指被调用的过程与调用过程处于不同的进程中，但同属于一台主机。
- 不同主机间的远程过程调用：指一台主机上的某个客户调用另外一台主机上的某个服务器的过程。

3. 什么是远程过程调用

RPC 是 Remote Procedure Call（远程过程调用）的缩写形式。Birrell 和 Nelson 在 1984 发表于 *ACM Transactions on Computer Systems* 的论文 *Implementing Remote Procedure Calls* 对 RPC 做了经典的诠释。RPC 是指计算机 A 上的进程，调用计算机 B 上的进程，其中 A 上的调用进程被挂起，而 B 上的被调用进程开始执行，当值返回 A 时，A 上的进程继续执行。调用方可以使用参数将信息传送给被调用方，而后可以通过传回的结果得到信息。这一过程对于开发人员来说是透明的。

远程过程调用采用客户机/服务器（C/S）模式。请求程序就是一个客户机，服务提供程序就是一台服务器。和常规或本地过程调用一样，远程过程调用是同步操作，在远程过程结果返回之前，需要暂时中止请求程序。使用相同地址空间的低权进程或低权线程允许同时运行多个远程过程调用。

图 1-10 描述了并发环境下 RPC 的调用过程。



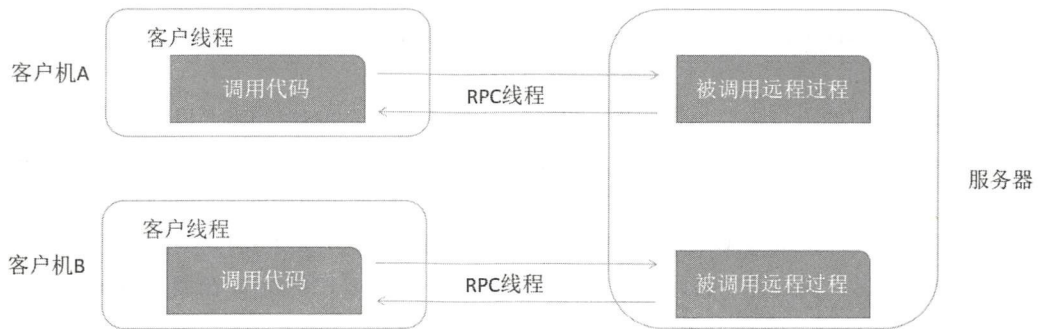


图 1-10 并发环境下 RPC 的调用过程

4. RPC 的基本操作

让我们看看本地过程调用是如何实现的。考虑下面的 C 语言的调用：

```
count = read(fd, buf, nbytes);
```

`fd` 为一个整型数，表示一个文件。`buf` 为一个字符数组，用于存储读入的数据。`nbytes` 为另一个整型数，用于记录实际读入的字节数。如果该调用位于主程序中，那么在调用之前堆栈的状态如图 1-11 (a) 所示。为了进行调用，调用方首先把参数反序压入堆栈，即最后一个参数先压入，如图 1-11 (b) 所示。在 `read` 操作运行完毕后，它将返回值放在某个寄存器中，移出返回地址，并将控制权交回给调用方。调用方随后将参数从堆栈中移出，使堆栈还原到最初的状态。

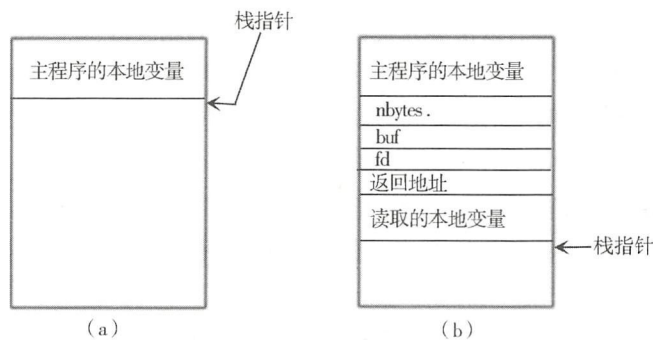


图 1-11 过程调用中的参数传递

RPC 背后的思想是尽量使远程过程调用具有与本地调用相同的形式。假设程序需要从某个文件中读取数据，程序员将在代码中执行 `read` 调用来取得数据。在传统的系统中，`read` 例程由链接器从库中提取出来，然后链接器再将它插入目标程序中。`read` 过程是一个短过程，一般通过执行一个等效的 `read` 系统调用来实现，即 `read` 过程是一个位于用户代码与本地操作系统之间的接口。

虽然 `read` 中执行了系统调用，但它本身依然是通过将参数压入堆栈的常规方式实现调用的。如图 1-11（b）所示，程序员并不知道 `read` 干了什么。

RPC 通过类似的途径来获得透明性。当 `read` 实际上是一个远程过程时（比如在文件服务器所在的机器上运行的过程），库中就放入 `read` 的另外一个版本，称为客户存根（client stub）。这种版本的 `read` 过程同样遵循图 1-11（b）的调用次序，这点与原来的 `read` 过程相同。另一个相同点是其中也执行了本地操作系统调用。唯一不同的是，它不要求操作系统提供数据，而是将参数打包成消息，而后请求将此消息发送到服务器，如图 1-12 所示。在调用 `send` 后，客户存根调用 `receive` 过程，随即阻塞自己，直到收到响应消息。

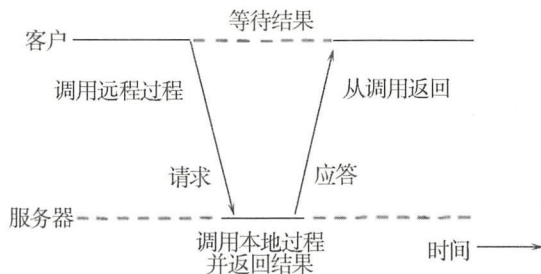


图 1-12 客户与服务器之间的 RPC 原理

当消息到达服务器时，服务器上的操作系统将它传递给服务器存根（server stub）。服务器存根是客户存根在服务器端的等价物，也是一段代码，用来将通过网络输入的请求转换为本地过程调用。服务器存根一般先调用 `receive`，然后被阻塞，等待消息输入。收到消息后，服务器将参数从消息中提取出来，然后以常规方式调用服务器上的相应过程。从服务器的角度看，过程好像是由客户直接调用的一样：参数和返回地址都位于堆栈中，一切都很正常。服务器执行所要求的操作，随后将得到的结果以常规的方式返回给调用方。以 `read` 为例，服务器将用数据填充 `read` 中第二个参数指向的缓存区，该缓存区是服务器存根内部的。

调用完后，服务器存根要将控制权交回给客户发出调用的过程，它将结果（缓存区）打包成消息，随后调用 `send` 将结果返回给客户。事后，服务器存根一般会再次调用 `receive`，等待下一个输入的请求。

客户端接收到消息后，客户操作系统发现该消息属于某个客户进程（实际上该进程是客户存根，只是操作系统无法区分二者）。操作系统将消息复制到相应的缓存区中，随后解除对客户进程的阻塞。客户存根检查该消息，将结果提取出来并复制给调用者，而后以通常的方式返回。当调用者在 `read` 调用进行完毕后重新获得控制权时，它所知道的唯一事情就是已经得到了所需的数据。它不知道操作是在本地操作系统中进行的，还是远程完成的。

在整个方法中，客户端可以简单地忽略不关心的内容。客户端涉及的操作只是执行普通的（本地）过程调用来访问远程服务，它并不需要直接调用 `send` 和 `receive`。消息传递的所有细节

都隐藏在双方的库过程中，就像传统库隐藏了执行实际系统调用的细节一样。

概括来说，远程过程调用包含如下步骤：

- (1) 客户过程以正常的方式调用客户存根。
- (2) 客户存根生成一个消息，然后调用本地操作系统。
- (3) 客户端操作系统将消息发送给远程操作系统。
- (4) 远程操作系统将消息交给服务器存根。
- (5) 服务器存根将参数提取出来，而后调用服务器。
- (6) 服务器执行要求的操作，操作完成后将结果返回服务器存根。
- (7) 服务器存根将结果打包成一个消息，而后调用本地操作系统。
- (8) 服务器操作系统将含有结果的消息发送给客户端操作系统。
- (9) 客户端操作系统将消息交给客户存根。
- (10) 客户存根将结果从消息中提取出来，返回调用它的客户存根。

以上步骤客户过程将客户存根发出的本地调用转换成对服务器过程的本地调用，而客户端和服务器都不会意识到中间步骤的存在。

RPC 的主要好处是双重的。首先，程序员可以使用过程调用语义来调用远程函数并获取响应。其次，简化了编写分布式应用程序的难度，因为 RPC 隐藏了所有的网络代码存根函数。应用程序不必担心一些细节，比如 socket、端口号，以及数据的转换和解析。在 OSI 参考模型中，RPC 跨越了会话层和表示层。

5. 实现远程过程调用

要实现远程过程调用，需要考虑以下几个问题。

1) 如何传递参数

传递值参数

传递值参数比较简单，图 1-13 是一个简单的通过 RPC 进行远程计算的例子。其中，远程过程 $\text{add}(i,j)$ 有两个参数 i 和 j ，其结果是返回 i 和 j 的算术和。

通过 RPC 进行远程计算的步骤如下：

- (1) 将参数放入消息中，并在消息中添加要调用的过程的名称或编码。
- (2) 消息到达服务器后，服务器存根对该消息进行分析，以判明需要调用哪个过程，随后执行相应的调用。
- (3) 服务器运行完毕，服务器存根将服务器得到的结果打包成消息返回客户存根，客户存根将结果从消息中提取出来，把结果值返回客户端。

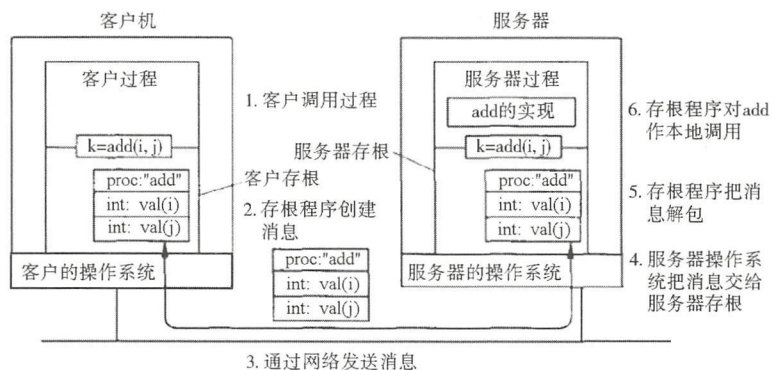


图 1-13 通过 RPC 进行远程计算的步骤

当然，这里只是做了简单的演示，在实际的分布式系统中，还需要考虑其他情况，因为不同的机器对于数字、字符和其他类型的数据项的表示方式常有差异，比如，整数型就有 Big Endian 和 Little Endian 之分。

传递引用参数

传递引用参数相对来说比较困难。单纯传递参数的引用（也包含指针）是完全没有意义的，因为引用地址传递给远程计算机，其指向的内存位置可能跟远程系统上的内存位置完全不同。如果你想支持传递引用参数，则必须发送参数的副本，将它们放置在远程系统内存中，向它们传递一个指向服务器函数的指针，然后将对象发送回客户端，复制它的引用。如果远程过程调用必须支持引用复杂的结构，比如树和链表，则它们需要将结构复制到一个无指针的表示里面（比如，一个扁平的树），并传输到远程系统来重建数据结构。

2) 如何表示数据

在本地系统中不存在数据不相容的问题，因为数据格式总是相同的。而在分布式系统中，不同的远程机器上可能有不同的字节顺序、不同大小的整数，以及不同的浮点表示。对于 RPC，如果想与异构系统通信，就需要一个“标准”来对所有数据类型进行编码，并可以作为参数传递。例如，ONC RPC 使用 XDR (eXternal Data Representation) 格式。这些数据表示格式可以使用隐式或显式类型。隐式类型指只传递值，而不传递变量的名称或类型。常见的例子是 ONC RPC 的 XDR 和 DCE RPC 的 NDR。显式类型指需要传递每个字段的类型和值。常见的例子是 ISO 标准 ASN.1 (Abstract Syntax Notation)、JSON (JavaScript Object Notation)、Google Protocol Buffers，以及各种基于 XML 的数据表示格式。

3) 如何选用传输协议

有些实现只允许使用一个协议（例如，TCP）。大多数 RPC 实现支持几个协议，并允许用户选择。

4) 出错时会发生什么

相比于本地过程调用，远程过程调用出错的机会更多。由于本地过程调用没有过程调用失败的概念，项目使用远程过程调用必须准备测试远程过程调用的失败或捕获异常。

5) 远程调用的语义是什么

调用一个普通的过程语义很简单：当我们调用时，过程被执行。远程过程完全一次性调用成功是非常难以实现的。执行远程过程可以有如下结果：

- 如果服务器崩溃或进程在运行服务器代码之前就死了，那么远程过程会被执行 0 次。
- 如果一切工作正常，则远程过程会被执行 1 次。
- 如果服务器返回服务器存根后在发送响应前就崩溃了，则远程过程会被执行 1 次或多次。客户端收不到返回的响应，可以决定再试一次，因此会出现多次执行函数的情况。如果没有则再试一次，函数执行 1 次。
- 如果客户机超时和重新传输，那么远程过程会被执行多次。也有可能原始请求延迟了，两者都可能执行或不执行。

RPC 系统通常会提供至少一次或最多一次的语义，或者在两者之间选择。如果需要了解应用程序的性质和远程过程的功能是否安全，则可以通过多次调用同一个函数来验证。如果一个函数可以运行任何次而不影响结果，则这是幂等（idempotent）函数，如每天的时间、数学函数、读取静态数据等。否则，它是一个非幂等（nonidempotent）函数，如添加或修改一个文件。

6) 远程调用的性能怎么样

毫无疑问，一个远程过程调用将比常规的本地过程调用慢得多，因为产生了额外的步骤及网络传输本身存在延迟。然而，这并不应该阻止我们使用远程过程调用。

7) 远程调用安全吗

使用 RPC，我们必须关注各种安全问题：

- 客户端发送消息到远程过程，这个过程是可信的吗？
- 客户端发送消息到远程计算机，这个远程机器是可信的吗？
- 服务器如何验证接收的消息来自合法的客户端？服务器如何识别客户端？
- 消息在网络中传播时如何防止被其他进程嗅探？
- 如何防止消息在客户端和服务器的网络传播中被其他进程拦截和修改？
- 协议能防止重播攻击吗？
- 如何防止消息在网络传播中被意外损坏或截断？

6. 远程过程调用的优点

远程过程调用有诸多优点：

- 不必担心传输地址问题。服务器可以绑定到任何可用的端口，然后用 RPC 名称服务来注册端口。客户端将通过该名称服务来找到对应的端口号所需的程序。而这一切对于程序员来说是透明的。
- 系统可以独立于传输提供者。自动生成服务器存根使其可以在系统上的任何一个传输提供者上可用，包括 TCP 和 UDP，而这些，客户端是可以动态选择的。当代码发送以后，接收消息是自动生成的，而不需要额外的代码。
- 应用程序在客户端只需要知道一个传输地址——名称服务，负责告诉应用程序去哪里连接服务器函数集。
- 使用函数调用模型来代替 socket 的发送/接收（读/写）接口。用户不需要处理参数的解析。

7. 远程过程调用 API

任何 RPC 实现都需要提供一组支持库。

- **名称服务操作：**注册和查找绑定信息（端口、机器）。允许一个应用程序使用动态端口（操作系统分配的）。
- **绑定操作：**使用适当的协议建立客户机/服务器通信（建立通信端点）。
- **终端操作：**注册端点信息（协议、端口号、机器名）到名称服务并监听过程调用请求。这些函数通常被自动生成的主程序——服务器存根（骨架）所调用。
- **安全操作：**系统应该提供机制来保证客户端和服务器之间能够相互验证，为两者提供一个安全的通信通道。
- **国际化操作（可能）：**目前，有一小部分 RPC 包支持转换包括时间格式、货币格式和特定语言的字符串的功能。
- **封送处理/数据转换操作：**函数将数据序列化为一个普通的的字节数组，通过网络进行传递，并能够重建。
- **存根内存管理和垃圾收集：**存根可能需要分配内存来存储参数，特别是模拟引用传递语义。RPC 包需要分配和清理任何这样的内存。它们也可能需要为创建网络缓存区而分配内存。RPC 包支持对象，RPC 系统需要跟踪远程客户端是否仍有引用对象或一个对象是否可以删除。
- **程序标识操作：**允许应用程序访问（或处理）RPC 接口集的标识符，这样服务器提供的接口集可以被用来交流和使用。
- **对象和函数的标识操作：**允许将远程函数或远程对象的引用传递给其他进程（并不是所有的 RPC 系统都支持）。

所以，判断一种通信方式是否是 RPC，就看它是否提供了上述的 API。

8. 远程过程调用发展历程

- **第一代 RPC：**Sun 公司是第一个提供商业化 RPC 库和 RPC 编译器的。在 20 世纪 80 年代中期 Sun 计算机提供 RPC，并在 Sun Network File System (NFS) 上得到支持。该协议被以 Sun 和 AT&T 为首的 Open Network Computing (开放网络计算) 作为一个标准来推动。这是一个非常轻量级的 RPC 系统，可在大多数 POSIX 和类 POSIX 操作系统中使用，包括 Linux、SunOS、OS X 和各种发布版本的 BSD。这样的系统被称为 Sun RPC 或 ONC RPC。该阶段的代表产品还有 DCE RPC。
- **第二代 RPC 支持对象：**面向对象的语言在 20 世纪 80 年代末兴起，很明显，当时的 Sun ONC 和 DCE RPC 系统都没有提供任何支持，诸如从远程类实例化远程对象、跟踪对象的实例或提供支持多态性。现有的 RPC 机制虽然可以运作，但它们仍然不支持自动、透明的面向对象编程技术。该阶段的主要产品有微软 DCOM (COM+)、CORBA 和 Java RMI。
- **第三代 RPC 及 Web Services：**传统 RPC 解决方案可以工作在互联网上，但问题是，它们通常严重依赖动态端口分配，往往要进行额外的防火墙配置。Web Services 成为一组协议，允许服务被发布、发现，并用于技术无关的形式，即服务不应该依赖于客户的语言、操作系统或机器架构。该阶段的代表产品有 XML-RPC、SOAP、Microsoft .NET Remoting 和 JAX-WS 等。

1.3.3 面向消息的通信

远程过程调用有助于隐藏分布式系统中的通信细节，也就是说增强了访问透明性。但这种机制并不一定适合所有场景，特别是当无法保证发出请求时接收端一定正在执行的情况下，就必须有其他的通信服务。同时 RPC 的同步特性也会造成客户在发出的请求得到处理之前就被阻塞了，因而有时也需要采取其他的办法。而面向消息的通信就解决了上面提到的种种问题。

面向消息的通信一般由消息队列系统 (Message-Queuing System, MQ) 或面向消息的中间件 (Message-Oriented Middleware, MOM) 提供高效可靠的消息传递机制来进行平台无关的数据交流，并可基于数据通信进行分布系统的集成。通过提供消息传递和消息排队模型，可在分布环境下扩展进程间的通信，并支持多种通信协议、语言、应用程序、硬件和软件平台。

通过使用 MQ 或 MOM，通信双方的程序 (称其为消息客户程序) 可以在不同的时间运行，程序不在网络上直接通话，而是间接地将消息放入 MQ 或 MOM 服务器的消息机制中。因为程序间没有直接的联系，所以它们不必同时运行：当消息放入适当的队列时，目标程序不需要正在运行；即使目标程序在运行，也不意味着要立即处理该消息。

消息客户程序之间通过将消息放入消息队列或从消息队列中取出消息来进行通信。客户程序不直接与其他程序通信，避免了网络通信的复杂性。消息队列和网络通信的维护工作由 MQ 或 MOM 完成。

常见的 MQ 或 MOM 产品有 Java Message Service、Apache ActiveMQ、Apache RocketMQ、RabbitMQ、Apache Kafka 等，这些产品在第 3 章中详细讲解。

Java Message Service (JMS)

Java Message Service (JMS) API 是一个 Java 面向消息中间件的 API，用于两个或多个客户端之间发送消息。

JMS 的目标包括：

- 包含实现复杂企业应用所需要的功能特性；
- 定义企业消息概念和功能的一组通用集合；
- 最小化这些 Java 程序员必须学习以使用企业消息产品的概念集合；
- 最大化消息应用的可移植性。

JMS 支持企业消息产品提供两种主要的消息风格：

- 点对点（Point-to-Point, PTP）消息风格——允许客户端通过一个叫“队列（queue）”的中间抽象发送一个消息给另一个客户端。发送消息的客户端将一个消息发送到指定的队列中，接收消息的客户端从这个队列中抽取消息。
- 发布订阅（Publish/Subscribe, Pub/Sub）消息风格——允许一个客户端通过一个叫“主题（topic）”的中间抽象发送一个消息给多个客户端。发送消息的客户端将一个消息发布到指定的主题中，然后这个消息将被投递到所有订阅了这个主题的客户端。

JMS API

由于历史的原因，JMS 提供了四组用于发送和接收消息的接口。

- JMS1.0 定义了两组特定领域相关的 API，一组用于点对点的消息处理（queue），另一组用于发布订阅的消息处理（topic）。尽管由于向后兼容的理由，这些接口一直被保留在 JMS 中，但是在以后的 API 中应该考虑被废弃掉。
- JMS1.1 引入了新的统一的一组 API，可以同时用于点对点和发布订阅消息模式。这也被称作标准（standard）API。
- JMS2.0 引入了一组简化 API，它拥有标准 API 的全部特性，同时接口更少、使用更方便。

以上每组 API 提供一组不同的接口集合，用于连接 JMS 提供者、发送和接收消息。因此，它们共享一组代表消息、消息目的地和其他各方面功能特性的通用接口。

下面是使用标准 API 来发送信息的例子：

```
@Resource(lookup = "jms/connectionFactory ")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public void sendMessageOld (String payload) throws JMSEException{
    try (Connection connection = connectionFactory.createConnection()) {
        Session session = connection.createSession();
        MessageProducer messageProducer =
            session.createProducer(inboundQueue);
        TextMessage textMessage =
            session.createTextMessage(payload);
        messageProducer.send(textMessage);
    }
}
```

下面是使用简化 API 来发送信息的例子：

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public void sendMessageNew (String payload) {
    try (MessagingContext context = connectionFactory.createMessagingContext();){
        context.send(inboundQueue,payload);
    }
}
```

所有的接口都在 `javax.jms` 包中。

更多有关 JMS 的规范可以在线查阅 <https://java.net/projects/jms-spec/pages/Home>。

1.4 一致性

分布式系统的一个重要问题是数据的复制。对数据进行复制一般是为了增强系统的可靠性和提高性能。举例来说，当一个数据库的副本被破坏以后，系统只需要转换到其他副本就能继续运行。另外一个例子，当访问单一服务器管理的数据的进程数不断增加时，系统就需要对服务器的数量进行扩充，此时，对服务器进行复制，随后让它们分担工作负荷，就可以提高性能。

但复制数据同时也带来了一个难点，那就是如何保持各个副本数据的一致性。换句话说，更新其中任意一个副本时，必须确保同时更新其他副本；否则，数据的各个副本将不再相同（数据不一致）。本节就来探讨如何实现数据的一致性。

1.4.1 以数据为中心的一致性模型

一致性模型实质上是进程和数据存储之间的一个约定。在正常情况下，在一个数据项上执行读操作时，它期待该操作返回的是该数据在其最后一次写操作之后的结果。在没有全局时钟的情况下，很难精确地定义哪次写操作是最后一次写操作，于是就产生了一系列用其他方式定义的一致性模型。

1. 严格一致性（Strict Consistency）

任意读操作都要读到最新的写的结果。严格一致性是限制性最强的模型，依赖于绝对的全局时钟，但是在分布式系统中实现这种模型的代价太大，所以在实际系统中的运用有限，基本上不可能做到。

2. 持续一致性（Continuous Consistency）

有多种不同的方法来为应用程序指定它们能容忍哪些不一致性，其中有一种通用的方法，它定义了区分不一致性的三个互相独立的坐标轴：副本之间的数值偏差、副本之间的新旧程度偏差，以及更新操作顺序的偏差。这些偏差形成了持续一致性的范围。

数值偏差可以这样理解：已应用于其他的副本，但还没有应用于给定副本的更新数目。比如，Web 缓存可能还没有得到 Web 服务器执行的一批操作。

新旧程度偏差与副本最近一次的更新有关。对于某些应用，只要副本提供的数据不是很旧，都是可以容忍的，比如，天气预报通常会滞后一段时间。

更新操作顺序的偏差是指，只要可以界定副本之间的差异，就允许不同的副本采用不同的更新顺序。

3. 顺序一致性 (Sequential Consistency)

任何执行结果都是相同的，就好像所有进程对数据存储的读或写操作是按某种序列顺序执行的一样，并且每个进程的操作按照程序所制定的顺序出现在这个序列中。

也就是说，任何读或写操作的交叉都是可接受的，但是所有进程都看到相同的操作交叉。

4. 因果一致性 (Casual Consistency)

所有进程必须以相同的顺序看到具有潜在因果关系的写操作。不同机器上的进程可以以不同的顺序看到并发的写操作。

假设 P1 和 P2 是有因果关系的两个进程，如果 P2 的写操作依赖于 P1 的写操作，那么 P1 和 P2 对 x 的修改顺序，在 P3 和 P4 看来一定是一样的。但如果 P1 和 P2 没有关系，那么 P1 和 P2 对 x 的修改顺序，在 P3 和 P4 看来可以是不一样的。

相比顺序一致性，因果一致性去掉了那些没有联系的操作需达成一致顺序观点的要求，只保留了那些必要的顺序（有因果关系）。

5. 入口一致性 (Entry Consistency)

入口一致性其实也就是对每个共享的数据定义一个同步变量（锁）。当然，没有进行同步就进行读操作，是不能保证结果正确的。

1.4.2 以客户为中心的一致性

以客户为中心的一致性是指从用户的视角来看数据是一致的。客户只关心数据最终是否一致。

只要保证同一个用户访问的数据是一致的就可以了。如果用户只是访问一个副本，则很好实现，否则就需要一定的策略了。当没有更多的更新时，要保证当前的更新最终会传播到所有副本上。著名的例子有 DNS 系统、万维网。

最终一致性需要注意一个典型的问题，即当客户访问不同的副本时，问题就出现了。更具体的例子比如，作者在博客上更改了一篇博文的内容，在 A 地的用户先访问到最新的内容，而 B 地由于离博客服务器远，读者看到的还是原先的内容。

对于最终一致性的数据存储而言，这个示例很有代表性。问题是由用户有时可能对不同的副本进行操作引起的。以客户为中心的一致性分为如下几大类。

1. 单调读一致性 (Monotonic-read Consistency)

当进程从一个地方读出数据 x ，那么以后再读到的 x 应该是和当前 x 相同或比当前更新的版本。也就是说，如果进程迁移到了别的位置，那么对 x 的更新应该比进程先到达。

以分布式邮件数据库系统为例。每个用户的邮箱可能分布式地复制在多台机器上。邮件可能被插入任何一个位置的邮箱。但是，数据更新是以一种懒惰的方式传播的。假设用户在杭州读取了他的邮件（假定只读取邮件不会影响其他邮箱，也就是说，消息不会被删除，甚至不会被标记为已读），当用户飞到惠州后，单调读一致性可以保证当他在惠州打开他的邮箱时，邮箱中仍然有杭州邮箱里的那些消息。

2. 单调写一致性 (Monotonic-write Consistency)

跟单调读相似，如果一个进程写一个数据 x ，那么它在本地或迁移到别的地方再进行写操作的时候，原来的写操作必须先传播到这个位置。也就是说，进程要在任何地方至少和上一次写一样新的数据。

3. 读写一致性 (Read-your-writes Consistency)

读写一致性指一个进程对于数据 x 的写操作，进程无论到任何副本上都应该能被后续读操作看到这个写操作的影响，也就是看到写操作的影响或更新的值。

也就是说，写操作总是在同一个进程执行的后续读操作之前完成的，而不管这个后续读操作发生在什么位置。

4. 写读一致性 (Writes-follow-reads Consistency)

顾名思义，写读一致性就是在读操作后面的写操作基于至少跟上一次读出来一样新的值。也就是说，如果进程在地点 1 读了 x ，那么在地点 2 要写 x 的副本的话，至少写的时候应该基于和地点 1 读出的一样新的值。

举个例子，用户先读了文章 A，然后他回复了一篇文章 B。为了满足读写一致性，B 被写入任何副本之前，需要保证 A 必须已经被写入那个副本。即，当原文章存储在某个本地副本上时，该文章的回应文章才能被存储到这个本地副本上。

1.5 容错性

集中式系统中任何一个组件的故障，都会导致整个系统无法正常使用。这就是为什么集中式系统的主机往往要求有比较高的性能。而分布式系统区别于集中式系统的一个特性是它容许部分失效。

分布式系统设计中的一个重要目标是以这样的方式构建系统：它可以从部分失效中自动恢复，而且不会严重地影响整体性能。特别是当故障发生时，分布式系统应该在进行恢复的同时继续以可接受的方式进行操作，也就是说，它应该能容忍错误，在发生错误时在某种程度上可以继续操作。

1.5.1 基本概念

容错往往与可靠的系统紧密相关，而可靠的系统需要满足以下要求。

- **可用性 (Availability)**：用来描述系统在给定时刻可以正确地工作。
- **可靠性 (Reliability)**：指系统可以无故障地连续运行。与可用性相反，可靠性是根据时间间隔而不是任何时刻来进行定义的。如果系统在每小时中崩溃的时间为 1ms，那么它的可用性就超过 99.9999%，但它还是高度不可靠的。与之相反，如果一个系统从来不崩溃，但是要在每年 8 月停机两个星期，那么它是高度可靠的，但是它的可用性只有 98%。因此，这两种属性并不相同。
- **安全性 (Safety)**：指系统在偶然出现故障的情况下能正确操作而不会造成任何灾难。
- **可维护性 (Maintainability)**：发生故障的系统被恢复的难易程度。

容错，意味着即使系统发生了故障，还能正常提供服务。

1.5.2 故障分类

故障通常被分为三类。

- **暂时故障 (Transient Fault)**：只发生一次，然后就消失了，不再重现该故障。
- **间歇故障 (Intermittent Fault)**：发生，消失不见，而后再次发生，如此反复进行。
- **持久故障 (Permanent Fault)**：那些直到故障组件被修复之前持续存在的故障。

分布式系统中的典型故障模式可以分为以下几种。

- **崩溃性故障 (Crash Failure)**：服务器停机，但是在停机之前工作正常。
- **遗漏性故障 (Omission Failure)**：服务器不能响应到来的请求。可以细分为服务器不能接收到来的消息，以及服务器不能发送消息。
- **定时性故障 (Timing Failure)**：服务器对请求响应得过快或者过慢。
- **响应性故障 (Response Failure)**：服务器对请求以错误的方式进行了响应。
- **任意性故障 (Arbitrary Failure)**：服务器可能在任意的时间产生任意类型的故障。其中，任意性故障是最严重的故障，也称为拜占庭故障 (Byzantine Failure)。当发生故障时，服务器可能产生它从来没有产生过的输出，但是又不能检测出错误。更坏的情况是，发生故障的服务器与其他服务器共同工作来产生恶意的错误结果。Windows 系统“蓝屏”正是为了尽可能避免这种情况而设计的。这种情况也说明了为什么谈到可靠系统时安全被认为是一个重要的需求。

随意性故障与崩溃性故障紧密相关。崩溃性故障的一个典型的例子就是操作系统崩溃，此时的解决办法只有一个：重新启动。与人们的期望相反，PC 经常遭遇崩溃性故障，最终导致设计者把复位按钮从机箱背后移到前面。

1.5.3 使用冗余来掩盖故障

如果系统是容错的，那么它能做的最好的事情就是对其他进程隐藏故障。关键技术是使用冗余来掩盖故障。有三种可能：信息冗余、时间冗余和物理冗余。

在信息冗余中，添加额外的位可以使错乱的位恢复正常。例如，可以在传输的数据中添加一段 Hamming 码来从传输线路上的噪声中恢复数据。可以利用信息冗余进行错误检测和纠正。

在时间冗余中，执行一个动作，如果需要就再次执行。使用事务就是这种方法的一个例子。如果一个事务中止，那么它就可以无害地重新执行。当发生临时性或间歇性的错误时，时间冗余特别有用。TCP/IP 协议中的重传机制是另外一个例子。

在物理冗余中，通过添加额外的装备或进程使系统作为一个整体来容忍部分组件的失效或故障成为可能。物理冗余可以在硬件上也可以在软件上进行。其中，一种著名的设计是 TMR（Triple Modular Redundancy，三倍模块冗余）。在包括 TMR 的系统中，每个关键模块中的部件都被复制了三份，采用多数表决的方法，确保当某些模块中的单个部件发生故障时，系统还可以正确地运行。

1.5.4 分布式提交

在分布式系统中，事务往往包含多个参与者的活动，单个参与者的活动是能够保证原子性的，而保证多个参与者之间原子性则需要通过两阶段提交或三阶段提交算法实现。

1. 两阶段提交

两阶段提交协议（Two-phase Commit Protocol, 2PC）的过程涉及协调者和参与者。协调者可以看作事务的发起者，同时是事务的一个参与者。对于一个分布式事务来说，一个事务是涉及多个参与者的。两阶段提交的具体过程如下。

第一阶段（准备阶段）

- 协调者节点向所有参与者节点询问是否可以执行提交操作（vote），并开始等待各参与者节点的响应。
- 参与者节点执行所有事务操作，并将 Undo 信息和 Redo 信息写入日志（注意：若成功这里其实每个参与者已经执行了事务操作）。

- 各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个“同意”消息；如果参与者节点的事务操作实际执行失败，则它返回一个“中止”消息。

第二阶段（提交阶段）

如果协调者收到了参与者的失败消息或超时，则直接给每个参与者发送回滚（rollback）消息；否则，发送提交（commit）消息；参与者根据协调者的指令执行提交或回滚操作，释放所有事务处理过程中使用的锁资源（注意：必须在最后阶段释放锁资源）。

- 当协调者节点从所有参与者节点处获得的相应消息都为“同意”时：
 - 协调者节点向所有参与者节点发出“正式提交（commit）”的请求。
 - 参与者节点正式完成操作，并释放在整个事务期间内占用的资源。
 - 参与者节点向协调者节点发送“完成”消息。
- 如果任一参与者节点在第一阶段返回的响应消息为“中止”，或者协调者节点在第一阶段的询问在超时之前无法获取所有参与者节点的响应消息时：
 - 协调者节点向所有参与者节点发出“回滚操作（rollback）”的请求。
 - 参与者节点利用之前写入的 Undo 信息执行回滚，并释放在整个事务期间占用的资源。
 - 参与者节点向协调者节点发送“回滚完成”消息。
 - 协调者节点收到所有参与者节点反馈的“回滚完成”消息后，取消事务。
 - 协调者节点收到所有参与者节点反馈的“完成”消息后，完成事务。

不管最后结果如何，第二阶段都会结束当前事务。

两段式提交协议的优缺点如下。

优点：原理简单，实现方便。

缺点：

- 同步阻塞问题。在执行过程中，所有参与节点都是事务阻塞型的。
- 单点故障。由于协调者的重要性，一旦协调者发生故障，参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者都还处于锁定事务资源的状态，而无法继续完成事务操作。
- 数据不一致。在阶段二中，当协调者向参与者发送 commit 请求之后，发生了局部网络异常，或者在发送 commit 请求过程中协调者发生了故障，这会导致只有一部分参与者接收到了 commit 请求。而在这部分参与者接收到 commit 请求之后就会执行 commit 操

作，但是其他部分未接收到 `commit` 请求的机器则无法执行事务提交操作。于是整个分布式系统便出现了数据不一致性的现象。

- 两阶段无法解决的问题：协调者再发出 `commit` 消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了，那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否已经被提交。

为了解决两阶段提交协议的种种问题，研究者在两阶段提交的基础上做了改进，提出了三阶段提交。

2. 三阶段提交

三阶段提交协议（Three-phase Commit Protocol, 3PC）是两阶段提交（2PC）的改进版本。与两阶段提交相比，三阶段提交有两点改动：

- 引入超时机制，同时在协调者和参与者中都引入超时机制。
- 在第一阶段和第二阶段中插入一个准备阶段，保证在最后提交阶段之前各参与节点的状态是一致的。

即 3PC 把 2PC 的准备阶段再次一分为二，这样三阶段提交就有 `CanCommit`、`PreCommit` 和 `DoCommit` 三个阶段。

CanCommit 阶段

`CanCommit` 阶段其实和 2PC 的准备阶段很像。协调者向参与者发送 `commit` 请求，参与者如果可以提交就返回 `yes` 响应，否则返回 `no` 响应。

- **事务询问：**协调者向参与者发送 `CanCommit` 请求，询问是否可以执行事务提交操作，然后开始等待参与者的响应。
- **响应反馈：**参与者接收到 `CanCommit` 请求之后，在正常情况下，如果其自身认为可以顺利执行事务，则返回 `yes` 响应，并进入预备状态，否则返回 `no` 响应。

PreCommit 阶段

协调者根据参与者的反应情况来决定是否执行事务的 `PreCommit` 操作。根据响应情况，有以下两种可能。

- 假如协调者从所有的参与者处获得的反馈都是 `yes` 响应，那么就会执行事务的预执行。
 - **发送预提交请求：**协调者向参与者发送 `PreCommit` 请求，并进入 `Prepared` 阶段。
 - **事务预提交：**参与者接收到 `PreCommit` 请求后，会执行事务操作，并将 `undo` 和 `redo` 信息记录到事务日志中。
 - **响应反馈：**如果参与者成功地执行了事务操作，则返回 `ACK` 响应，同时开始等待最终指令。

- 假如任何一个参与者向协调者发送了 no 响应，或者等待超时之后，协调者都没有接收到参与者的响应，那么就执行事务的中断操作。
- **发送中断请求：**协调者向所有参与者发送 abort 请求。
- **中断事务：**参与者接收到来自协调者的 abort 请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断操作。

doCommit 阶段

该阶段进行真正的事务提交，也可以分为以下两种情况。

- **执行提交**
 - **发送提交请求：**协调者接收到参与者发送的 ACK 响应，那么它将从预提交状态进入提交状态，并向所有参与者发送 doCommit 请求。
 - **事务提交：**参与者接收到 doCommit 请求之后，正式进行事务提交，并在完成事务提交之后释放所有事务资源。
 - **响应反馈：**事务提交完之后，向协调者发送 ACK 响应。
 - **完成事务：**协调者接收到所有参与者的 ACK 响应之后，完成事务。
- **中断事务：**协调者没有接收到参与者发送的 ACK 响应（可能是接收者发送的不是 ACK 响应，也可能响应超时），那么就会执行中断事务操作。
 - **发送中断请求：**协调者向所有参与者发送 abort 请求。
 - **事务回滚：**参与者接收到 abort 请求之后，利用其在阶段二记录的 undo 信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。
 - **反馈结果：**参与者完成事务回滚之后，向协调者发送 ACK 消息。
 - **中断事务：**协调者接收到参与者反馈的 ACK 消息之后，执行事务的中断操作。

在 doCommit 阶段，如果参与者无法及时接收到来自协调者的 doCommit 或 rebort 请求，则会在等待超时之后，继续进行事务的提交。即当进入第三阶段时，由于网络超时等原因，虽然参与者没有接收到 commit 或 abort 响应，事务仍然会提交。

三阶段提交不会一直持有事务资源并处于阻塞状态。但是这种机制也会导致数据一致性问题，因为，如果协调者发送的 abort 响应由于网络原因没有及时被参与者接收到，那么参与者在等待超时之后执行了 commit 操作，这样就和其他接到 abort 命令并执行回滚的参与者之间存在数据不一致的情况。

3. Paxos 算法

Paxos 算法是 Leslie Lamport 于 1990 年提出的一种基于消息传递且具有高度容错特性的一

致性算法。Paxos 算法目前在 Google 的 Chubby、MegaStore、Spanner 等系统中得到了应用，Hadoop 中的 ZooKeeper 也使用了 Paxos 算法。

在 Paxos 算法中有 4 种角色。

- Proposer: 提议者;
- Acceptor: 决策者;
- Client: 产生议题者;
- Learner: 最终决策学习者。

算法可以分两个阶段执行。

阶段 1

- Proposer: 选择一个议案编号 n ，向 Acceptor 的多数派发送编号也为 n 的 prepare 请求。
- Acceptor: 如果接收到的 prepare 请求的编号 n 大于它已经回应的任何 prepare 请求，则它就回应已经批准的编号最高的议案（如果有的话），并承诺不再回应任何编号小于 n 的议案。

阶段 2

- Proposer: 如果收到了多数 Acceptor 对 prepare 请求（编号为 n ）的回应，则它就向这些 Acceptor 发送议案 $\{n, v\}$ 的 accept 请求，其中 v 是所有回应中编号最高的议案的决议，或者是 Proposer 选择的值，如果响应中不包含议案，那么它就是任意值。
- Acceptor: 如果收到了议案 $\{n, v\}$ 的 accept 请求，则它就批准该议案，除非它已经回应了一个编号大于 n 的议案。
- Proposer 可以提出多个议案，只要它遵循上面的算法。它可以在任何时刻放弃一个议案（这不会破坏正确性，即使在议案被放弃后议案的请求或回应消息才到达目标）。如果其他 Proposer 已经开始提出更高编号的议案，那么最好能放弃当前的议案。因此，如果 Acceptor 忽略一个 prepare 或 accept 请求（因为已经收到了更高编号的 prepare 请求），则它应该告知 Proposer 放弃议案。这是一个性能优化，而不影响正确性。

该算法的详细描述可以在线参阅 <http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf>。

1.6 CAP 理论

在单机的数据库系统中，我们很容易就可以实现一套满足 ACID 特性的事务处理系统，事务的一致性不存在问题。但是在分布式系统中，由于数据分布在不同的主机节点上，对这些数

据进行分布式的事务处理具有非常大的挑战。CAP 理论的出现，让我们对于分布式事务的一致性有了另一种看法。

1.6.1 什么是 CAP 理论

在计算机科学理论中，CAP 理论（也称为 Brewer 定理）是由计算机科学家 Eric Brewer 在 2000 年提出的，其理论观点是，在分布式计算机系统中不可能同时提供以下三个保证。

- 一致性（Consistency）：所有节点同一时间看到的是相同的数据。
- 可用性（Availability）：不管是否成功，确保每一个请求都能接收到响应。
- 分区容错性（Partition Tolerance）：将系统任意分区后，在网络故障时，仍能操作。

CAP 理论如图 1-14 所示。

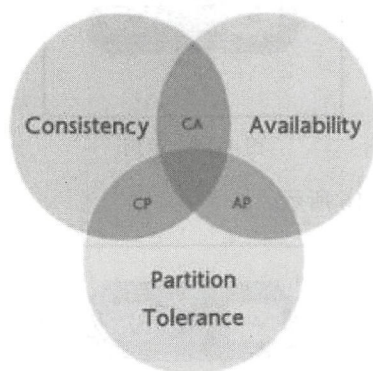


图 1-14 CAP 理论

在 2003 年的时候，Gilbert 和 Lynch 就正式证明了这三个特征确实是不可以兼得的。Gilbert 认为这里所说的一致性（Consistency）其实就是数据库系统中提到的 ACID 的另一种表述：一个用户请求要么成功、要么失败，不能处于中间状态（Atomic）；一旦一个事务完成，将来的所有事务都必须基于这个完成后的状态（Consistent）；未完成的事务不会互相影响（Isolated）；一旦一个事务完成，就是持久的（Durable）。对于可用性（Availability），其概念没有变化，指的是对于一个系统而言，所有的请求都应该“成功”并且收到“返回”。分区容错性（Partition Tolerance）指就是分布式系统的容错性。节点崩溃或者网络分片都不应该导致一个分布式系统停止服务。

1.6.2 为什么 CAP 只能三选二

下面举例说明为什么 CAP 只能三选二。

图 1-15 显示了在一个网络中， N_1 和 N_2 两个节点共享数据块 V ，其中有一个值 V_0 。运行在 N_1 上的 A 程序可以认为是安全、无 bug、可预测和可靠的。运行在 N_2 上的是 B 程序。在这个例子中，A 将写入 V 的新值，而 B 从 V 中读取值。

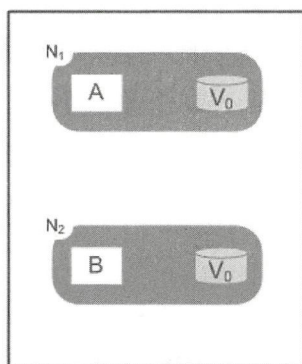


图 1-15 示例

系统预期执行的操作如图 1-16 所示。

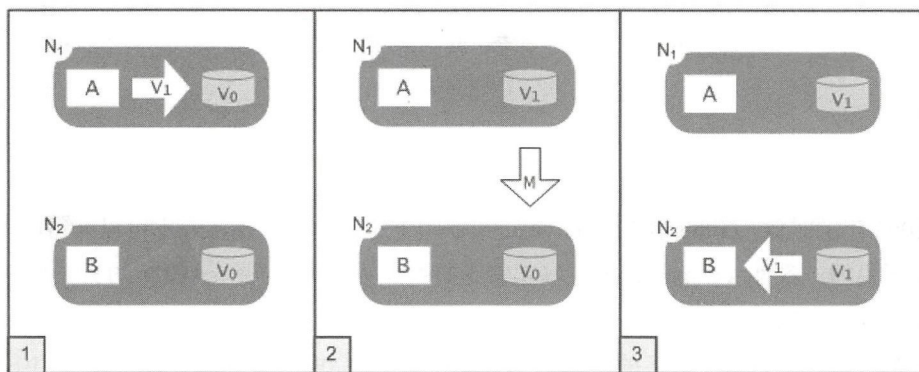


图 1-16 系统预期执行的操作

- (1) 写一个 V 的新值 V_1 。
- (2) 消息 (M) 从 N_1 更新 V 的副本到 N_2 。
- (3) 从 B 读取返回的 V_1 。

如果网络是分区的，当 N_1 到 N_2 的消息不能传递时，执行图 1-17 中的第三步，会出现虽然

N_2 能访问 V 的值（可用性），但其实与 N_1 的 V 的值已经不一致了（一致性）的情况。

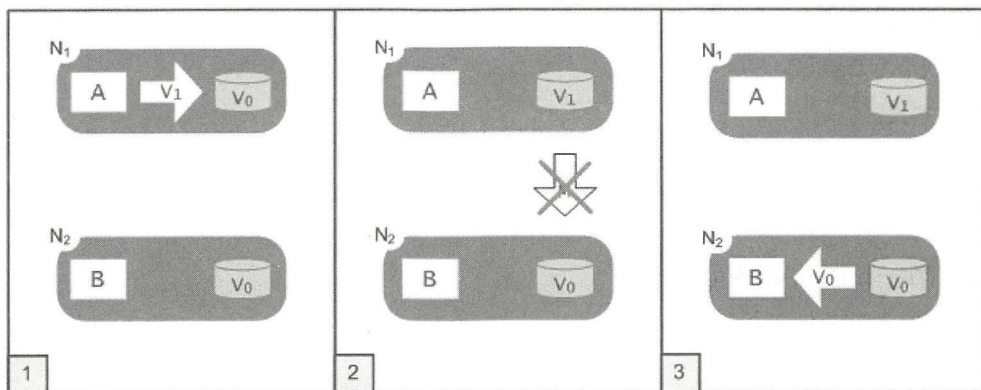


图 1-17 出现值不一致问题

1.6.3 CAP 常见模型

既然 CAP 理论已经证明了一致性、可用性、分区容错性三者不可能同时达成。那么在实际应用中，可以在其中的某一些方面来放松条件，从而达到妥协。下面是常见的三种 CAP 模型。

1. 牺牲分区（CA 模型）

牺牲分区容错性意味着把所有的机器搬到一台机器内部，或者放到一个“要死大家一起死”的机架上面（当然机架也可能部分失效），这明显违背了我们希望获得的可伸缩性。

CA 模型常见的例子：

- 单站点数据库；
- 集群数据库；
- LDAP；
- xFS 文件系统。

实现方式：

- 两阶段提交；
- 缓存验证协议。

2. 牺牲可用性（CP 模型）

牺牲可用性意味着一旦系统中出现分区这样的错误，则系统直接停止服务。

CP 模型常见的例子：

- 分布式数据库；
- 分布式锁定；
- 绝大部分协议。

实现方式：

- 悲观锁；
- 少数分区不可用。

3. 牺牲一致性（AP 模型）

AP 模型常见的例子：

- Coda；
- Web 缓存；
- DNS。

实现方式：

- 到期/租赁；
- 解决冲突；
- 乐观。

1.6.4 CAP 的意义

在构建系统时，应该根据具体的业务场景来权衡 CAP。比如，对于大多数互联网应用来说（如门户网站），因为机器数量庞大，部署节点分散，网络故障是常态，可用性是必须要保证的，所以只有舍弃一致性来保证服务的 AP。而对于银行等需要确保一致性的场景，通常会权衡 CA 和 CP 模型，CA 模型出现网络故障时完全不可用，CP 模型具备部分可用性。

1.6.5 CAP 最新发展

Eric Brewer在 2012 年发表文章²指出了CAP里面“三选二”的做法存在一定的误导性，主要体现在：

- 由于分区很少发生，在系统不存在分区的情况下没什么理由牺牲 C 或 A；
- C 与 A 之间的取舍可以在同一系统内以非常细小的粒度反复发生，而每一次的决策可

² 该文章可以在线查阅 <https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed>。

能因为具体的操作，乃至因为牵涉特定的数据或用户而有所不同；

- 这三种性质都可以在一定程度上衡量，并不是非黑即白的有或无。可用性显然是在 0% 到 100%之间连续变化的。一致性分很多级别，连分区也可以细分为不同含义，如系统内的不同部分对于是否存在分区可以有不一样的认知。

理解 CAP 理论最简单的方式是想象两个节点分处分区两侧。允许至少一个节点更新状态会导致数据不一致，即丧失了 C 性质。如果为了保证数据的一致性，将分区一侧的节点设置为不可用，那么又丧失了 A 性质。除非两个节点可以互相通信，才能既保证 C 又保证 A，但这又会导致丧失 P 性质。一般来说，跨区域的系统，架构师无法舍弃 P 性质，就只能在数据一致性和可用性上做一个艰难的选择。不确切地说，NoSQL 运动的主题其实是创造各种可用性优先、数据一致性其次的方案；而传统数据库坚守 ACID 特性，做的是相反的事情。

BASE

BASE (Basically Available、Soft state、Eventual consistency) 来自互联网电子商务领域的实践，它是基于 CAP 理论逐步演化而来的，核心思想是即便不能达到强一致性 (Strong consistency)，也可以根据应用特点采用适当的方式来达到最终一致性 (Eventual consistency) 的效果。BASE 是对 CAP 中 C 和 A 的延伸。BASE 的含义如下。

- Basically Available: 基本可用。
- Soft state: 软状态、柔性事务，即状态可以有一段时间的不同步。
- Eventual consistency: 最终一致性。

BASE 是反 ACID 的，它完全不同于 ACID 模型，通过牺牲强一致性获得基本可用性和柔性可靠性并要求达到最终一致性。

1.7 安全性

计算机的安全性通常包括两个部分：认证和访问控制。认证包括对有效用户身份的确认和识别。而访问控制则致力于避免对数据文件和系统资源的有害篡改。举例来说，在一个孤立、集中、单用户系统中（例如一台计算机），通过锁上存放该计算机的房间并将磁盘锁起来就能够实现安全性，只有拥有房间和磁盘钥匙的用户才能访问系统资源和文件——这就同时实现了认证和访问控制。因此，安全性实际上就相当于锁住计算机和房间的钥匙。

分布式系统的安全比集中式系统复杂得多，因为安全涉及整个系统，所以有关安全的任何设计缺陷都可能导致所有的安全措施无效。分布式系统是由各个子系统组成的，子系统之间也需要做身份识别；各子系统都是通过网络进行连接的，它们之间的安全通信也是需要保障的。

由于安全是一个复杂的话题，并不是本书所要阐述的重点，所以本书只会涉及分布式系统

安全方面的基本知识。

1.7.1 基本概念

1. 安全威胁、策略和机制

计算机系统的安全性与其可靠性密切相关。非正式地说，一个可靠的计算机系统是一个我们有理由信任其所提供的服务的系统。可靠性包括可用性、可信赖性、安全性和可维护性。如果我们要信任一个计算机系统，则还应该考虑机密性和完整性。考虑计算机系统中安全的另一种角度是我们试图保护该系统所提供的服务和数据不受到安全威胁，这类安全威胁包括4种。

- **窃听**：指一个未经授权的用户获得了对一项服务或数据的访问权限。窃听的一个典型实例是两人之间的通信被其他人偷听。窃听还发生在非法复制数据时。
- **中断**：指服务或数据变得难以获得、不能使用、被破坏等情况。从这个意义上说，服务拒绝攻击是一种安全威胁，它被归为中断类，一些人正是通过它恶意地使其他人不能访问服务。
- **修改**：包括对数据未经授权的改变或篡改一项服务以使其不再遵循其原始规范。修改的实例包括窃听，然后改变传输的数据，篡改数据库条目，以及改变一个程序使其秘密记录其用户的活动。
- **伪造**：指产生通常不存在的附加数据或活动的情况。例如，一个入侵者可能尝试向密码文件或数据库中添加数据，同样，有时可能通过重放先前发送过的消息来侵入一个系统。

仅仅声明系统应该能够保护其自身免受任何可能的安全威胁并不是实际建立一个安全系统的方式。首先需要的是一个安全需求的描述，也就是一个策略。安全策略准确地描述了系统中的实体能够采取的行为及禁止采取的行动。实体包括用户、服务、数据、机器等。制定安全策略之后，就可以集中考虑安全机制，策略通过该机制来实施。重要的机制包括：

- 加密；
- 身份验证；
- 授权；
- 审计。

加密将数据转换为一些攻击者不能理解的形式。身份验证用于检验用户、客户、服务器等所声明的身份。对一个客户进行身份验证之后，有必要检查是否授予客户执行该请求操作的权限。审计工具用于追踪各个客户的访问内容和访问方式。

2. 密码与数字签名

加密包括使用密钥对数据进行编码，从而使偷听者无法方便地阅读这些数据。经过加密的数据称为密文，原始的数据称为明文。从密文到明文的转换过程称为解密。

图 1-18 描述了 Alice 和 Bob 通过 Caesar 密码方式进行通信的过程，这里的 Alice 和 Bob 常用来表示密码通信的双方。

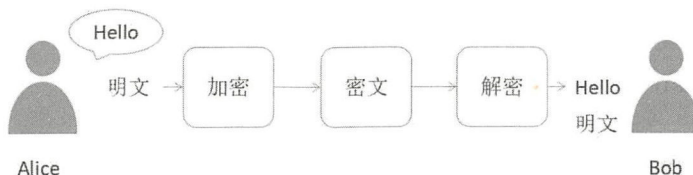


图 1-18 Caesar 密码通信过程

Caesar 密码又叫循环移位密码，它的加密方法就是将明文中的每个字母用字母表中该字母后的第 R 个字母来替换，从而达到加密的目的。

它的加密过程可以表示为下面的函数：

$$E(m) = (m+k) \bmod n$$

其中， m 为明文字母在字母表中的位置数； n 为字母表中的字母个数； k 为密钥； $E(m)$ 为密文字母在字母表中对应的位置数。例如，对于明文字母 H，其在字母表中的位置数为 8，则按照上式计算出来的密文为 L，计算过程如下：

$$E(8) = (8+4) \bmod 26 = 12$$

在上面的例子中，每个字母用其后面的第三个字母代替。

下面是一个用 C 语言实现的 Caesar 密码程序：

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char *caesar(const char *str,int offset)
{
    char *start,*ret_str;
    start = ret_str = (char *) malloc(strlen(str) + 1);
    for(;*str!='\0';str++,ret_str++)
    {
        if(*str>='A' && *str<='Z')
            *ret_str = 'A' + (*str - 'A' + offset) % 26;
        else if(*str>='a' && *str<='z')

```

```
        *ret_str = 'a' + (*str - 'a' + offset) % 26;
    else
        *ret_str = *str;
    }
    *ret_str = '\0';
    return (char *) start;
}

int main(void)
{
    printf("%s\n", "ABCDEFGHIJKLMNOPQRSTUVWXYZ");
    printf("%s\n", caesar("ABCDEFGHIJKLMNOPQRSTUVWXYZ", 3));
    return 0;
}
```

1.7.2 加密算法

评估一种加密算法的安全性常用的方法是判断该算法是否是计算安全的。如果利用可用资源进行系统分析后无法攻破系统，那么这种加密算法就是计算安全的。目前有两种常用的加密类型：私钥加密和公钥加密。除了加密整条消息，两种加密类型都可以用来对一个文档进行数字签名。当一个密钥越长时，密码被破解的难度就会越大，系统也就越安全。当然，密钥越长，本身加解密的成本也就越高。

1. 对称加密

对称加密：加密和解密算法都使用相同密钥的加密算法。具体如下：

$$E(p, k) = C ; D(C, k) = p$$

其中

- E = 加密算法；
- D = 解密算法；
- p = 明文（原始数据）；
- k = 加密密钥；
- C = 密文。

由于在加密和解密数据时使用同一个密钥，因此这个密钥必须保密，这样的加密称为对称密钥算法，或单密钥算法，或常规加密。很显然，对称算法的安全性依赖于密钥，泄露密钥就



意味着任何人都可以对他们发送或接收的消息解密，所以密钥的保密性对通信的安全性至关重要。

对称加密算法的特点是算法公开、计算量小、加密速度快、加密效率高。

对称加密算法的不足之处是，交易双方都使用同样的钥匙，安全性得不到保证。此外，每对用户每次使用对称加密算法时，都需要使用其他人不知道的唯一钥匙，这会使得发收信双方所拥有的钥匙数量呈几何级数增长，密钥管理成为用户的负担。对称加密算法在分布式网络系统上使用较为困难，主要是因为密钥管理困难，使用成本较高。与公开密钥加密算法比起来，对称加密算法能够提供加密和认证却缺乏签名功能，使用范围较小。

常见的对称加密算法有 DES、3DES、TDEA、Blowfish、RC2、RC4、RC5、IDEA、SKIPJACK、AES 等。

2. 使用对称密钥加密的数字签名

在通过网络发送数据的过程中，有两种对文档进行数字签名的基本方法。这里我们讨论第一种方法，利用私钥加密法。数字签名也称为**消息摘要**（Message Digest）或**数字摘要**（Digital Digest），它是一个对应唯一消息或文本的固定长度的值，它由一个单向 Hash 加密函数对消息进行作用而产生。如果消息在途中改变了，则接收者通过对收到的消息新产生的摘要与原摘要进行比较，就可以知道消息是否被改变了。因此消息摘要保证了消息的完整性。消息摘要采用单向 Hash 函数将需加密的明文“摘要”成一串 128bit 的密文，这一串密文也称为**数字指纹**（Finger Print），它有固定的长度，且不同的明文摘要成密文，其结果总是不同的，而同样的明文其摘要必定一致。这样这串摘要便可成为验证明文是否是“真身”的“指纹”了。有两种方法可以利用共享的私钥来计算摘要。最简单、快捷的方法是计算消息的哈希值，然后通过私钥对这个数值进行加密，再把消息和已加密的摘要一起发送。接收者可以再次计算消息摘要，对摘要进行加密，并与接收到的加密摘要进行比较。如果这两个加密摘要相同，就说明该文档没有被改动。第二种方法是将私钥应用到消息上，然后计算哈希值。

计算公式为：

$$D(M, K)$$

其中

- D 是摘要函数；
- M 是消息；
- K 是共享的私钥。

然后可以发布或分发这个文档。由于第三方并不知道私钥，而计算正确的摘要值恰恰需要它，因此消息摘要能够避免对摘要值自身的伪造。在这两种情况下，只有那些了解密钥的用户



才能验证其完整性，所有欺骗性的文档都可以很容易地被检验出来。

消息摘要算法有 MD2、MD4、MD5、SHA-1、SHA-256、RIPEMD128、RIPEMD160 等。

3. 非对称加密

非对称加密（也称为公钥加密）由两个密钥组成，包括公开密钥（public key，简称公钥）和私有密钥（private key，简称私钥）。如果信息使用公钥进行加密，那么使用对应的私钥可以解密这些信息，过程如下：

$$E(p, ku) = C ; D(C, kr) = p$$

其中

- E = 加密算法；
- D = 解密算法；
- p = 明文（原始数据）；
- ku = 公钥；
- kr = 私钥；
- C = 密文。

如果信息使用私钥进行加密，那么使用对应的公钥可以解密这些信息，过程如下：

$$E(p, kr) = C ; D(C, ku) = p$$

其中

- E = 加密算法；
- D = 解密算法；
- p = 明文（原始数据）；
- ku = 公钥；
- kr = 私钥；
- C = 密文。

不能使用加密所用的密钥来解密一个消息。而且，由一个密钥计算出另一个密钥从数学上来说是很困难的。私钥只有用户本人知道，并因此得名。公钥并不保密，可以通过公共列表服务获得，通常公钥是使用 X.509 实现的。公钥加密的想法最早是由 Diffie 和 Hellman 于 1976 年提出的。

非对称加密与对称加密相比，安全性更好。对称加密的通信双方使用相同的密钥，如果一方的密钥遭泄露，那么整个通信就会被破解。而非对称加密使用一对密钥，一个用来加密，另一个用来解密，而且公钥是公开的，密钥是自己保存的，不需要像对称加密那样在通信之前先





同步密钥。

非对称加密的缺点是加密和解密花费时间长、速度慢，只适合对少量数据进行加密。

在非对称加密中使用的主要算法有 RSA、Elgamal、背包算法、Rabin、D-H、ECC（椭圆曲线加密算法）等。

4. 使用公钥加密的数字签名

用于数字签名的公钥加密使用 RSA 算法。在这种方法中，发送者利用私钥通过摘要函数对整个数据文件（代价昂贵）或文件的签名进行加密。私钥匹配最主要的优点就是不存在密钥分发问题。这种方法假定你信任发布公钥的来源，然后接收者可以利用公钥来解密签名或文件，并验证它的来源和/或内容。由于公钥密码学的复杂性，只有正确的公钥才能够解密信息或摘要。最后，如果你要将消息发送给拥有已知公钥的用户，那么就可以使用接收者的公钥来加密消息或摘要，这样只有接收者才能够通过他们自己的私钥来验证其中的内容。

1.7.3 安全通道

1. SSL/TLS

SSL（Secure Sockets Layer，安全套接字层）是在网络上应用最广泛的加密协议实现。SSL 结合加密过程来提供网络的安全通信。

SSL 提供了一个安全的增强标准 TCP/IP 套接字用于网络通信协议。如表 1-1 所示，在标准 TCP/IP 协议栈的传输层和应用层之间添加了完全套接字层。SSL 的应用程序中最常用的是 Hypertext Transfer Protocol（HTTP，超文本传输协议），它是互联网网页协议。其他应用程序，如 Net News Transfer Protocol（NNTP，网络新闻传输协议）、Telnet、Lightweight Directory Access Protocol（LDAP，轻量级目录访问协议）、Interactive Message Access Protocol（IMAP，互动信息访问协议）和 File Transfer Protocol（FTP，文件传输协议），也可以使用 SSL。

注：目前还没有标准的安全的 FTP。

表 1-1 增强标准的 TCP/IP 层与协议

TCP/IP 层	协 议
Application Layer	HTTP、NNTP、Telnet、FTP 等
Secure Sockets Layer	SSL
Transport Layer	TCP
Internet Layer	IP

SSL 最初是由网景公司在 1994 年创立的，现在已经演变为一个标准。由国际标准组织





Internet Engineering Task Force(IETF)进行管理。之后 IETF 更名 SSL 为 Transport Layer Security (TLS, 传输层安全)，并在 1999 年 1 月发布了第一个规范，版本为 1.0。TLS 1.0 对于 SSL 的最新版本 3.0 版本是一个小的升级，两者差异非常微小。TLS 1.1 是在 2006 年 4 月发布的，TLS 1.2 在 2008 年 8 月发布。

2. SSL 握手过程

SSL 通过握手过程在客户端和服务端之间协商会话参数并建立会话。会话包含的主要参数有会话 ID、对方的证书、加密套件（密钥交换算法、数据加密算法和 MAC 算法等）及主密钥（master secret）。通过 SSL 会话传输的数据，都将采用该会话的主密钥和加密套件进行加密、计算 MAC 等。

在不同的情况下，SSL 的握手过程存在差异。下面将分别描述以下三种握手过程：

- 只验证服务器的 SSL 握手过程；
- 验证服务器和客户端的 SSL 握手过程；
- 恢复原有会话的 SSL 握手过程。

只验证服务器的 SSL 握手过程如图 1-19 所示。

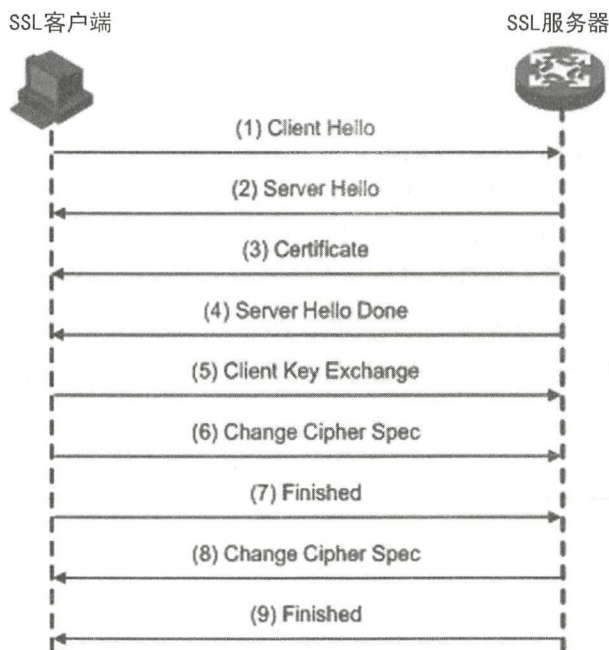


图 1-19 只验证服务器的 SSL 握手过程

如图 1-19 所示，在只需要验证 SSL 服务器身份、不需要验证 SSL 客户端身份时，SSL 的





握手过程如下。

(1) SSL 客户端通过 Client Hello 消息将它支持的 SSL 版本、加密算法、密钥交换算法、MAC 算法等信息发送给 SSL 服务器。

(2) SSL 服务器确定本次通信采用的 SSL 版本和加密套件，并通过 Server Hello 消息通知 SSL 客户端。如果 SSL 服务器允许 SSL 客户端在以后的通信中重用本次会话，则 SSL 服务器会为本次会话分配会话 ID，并通过 Server Hello 消息发送给 SSL 客户端。

(3) SSL 服务器将携带自己公钥信息的数字证书通过 Certificate 消息发送给 SSL 客户端。

(4) SSL 服务器发送 Server Hello Done 消息，通知 SSL 客户端版本和加密套件协商结束，开始进行密钥交换。

(5) SSL 客户端验证 SSL 服务器的证书合法后，利用证书中的公钥加密 SSL 客户端随机生成的 premaster secret，并通过 Client Key Exchange 消息发送给 SSL 服务器。

(6) SSL 客户端发送 Change Cipher Spec 消息，通知 SSL 服务器后续报文将采用协商好的密钥和加密套件进行加密和 MAC 计算。

(7) SSL 客户端计算已交互的握手消息（除 Change Cipher Spec 消息外所有已交互的消息）的 Hash 值，利用协商好的密钥和加密套件处理 Hash 值（计算并添加 MAC 值、加密等），并通过 Finished 消息发送给 SSL 服务器。SSL 服务器利用同样的方法计算已交互的握手消息的 Hash 值，并与 Finished 消息的解密结果比较，如果二者相同，且 MAC 值验证成功，则证明密钥和加密套件协商成功。

(8) 同样，SSL 服务器发送 Change Cipher Spec 消息，通知 SSL 客户端后续报文将采用协商好的密钥和加密套件进行加密和 MAC 计算。

(9) SSL 服务器计算已交互的握手消息的 Hash 值，利用协商好的密钥和加密套件处理 Hash 值（计算并添加 MAC 值、加密等），并通过 Finished 消息发送给 SSL 客户端。SSL 客户端利用同样的方法计算已交互的握手消息的 Hash 值，并与 Finished 消息的解密结果比较，如果二者相同，且 MAC 值验证成功，则证明密钥和加密套件协商成功。

SSL 客户端接收到 SSL 服务器发送的 Finished 消息后，如果解密成功，则可以判断 SSL 服务器是数字证书的拥有者，即 SSL 服务器身份验证成功，因为只有拥有私钥的 SSL 服务器才能从 Client Key Exchange 消息中解密得到 premaster secret，从而间接地实现 SSL 客户端对 SSL 服务器的身份验证。

验证服务器和客户端的 SSL 握手过程如图 1-20 所示。



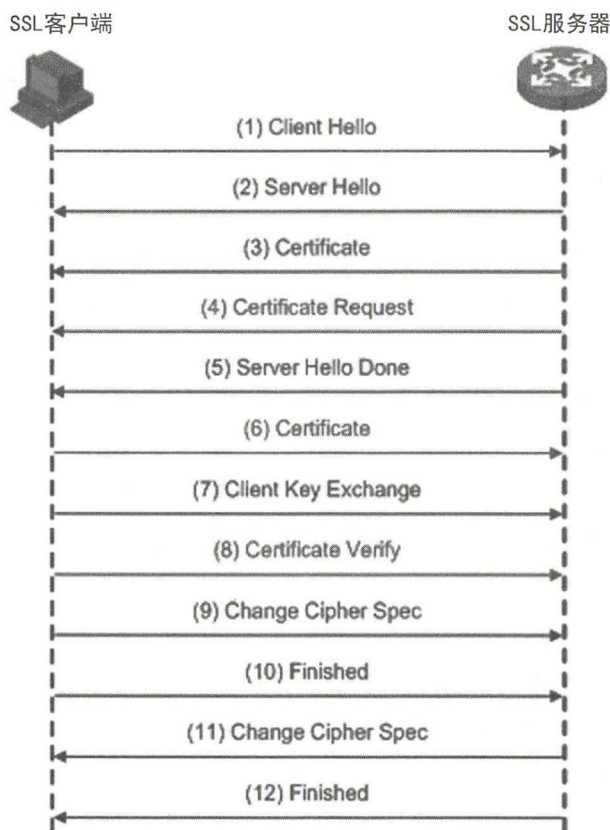


图 1-20 验证服务器和客户端的 SSL 握手过程

SSL 客户端的身份验证是可选的，由 SSL 服务器决定是否验证 SSL 客户端的身份。如图 1-20 中（4）、（6）、（8）部分所示，如果 SSL 服务器验证 SSL 客户端身份，则 SSL 服务器和 SSL 客户端除了交互“只验证服务器的 SSL 握手过程”中的消息协商密钥和加密套件，还需要进行以下操作。

（1）SSL 服务器发送 Certificate Request 消息，请求 SSL 客户端将其证书发送给 SSL 服务器。

（2）SSL 客户端通过 Certificate 消息将携带自己公钥的证书发送给 SSL 服务器。SSL 服务器验证该证书的合法性。

（3）SSL 客户端计算已交互的握手消息、主密钥的 Hash 值，利用自己的私钥对其进行加密，并通过 Certificate Verify 消息发送给 SSL 服务器。

（4）SSL 服务器计算已交互的握手消息、主密钥的 Hash 值，利用 SSL 客户端证书中的公钥解密 Certificate Verify 消息，并将解密结果与计算出的 Hash 值比较。如果二者相同，则 SSL





客户端身份验证成功。

恢复原有会话的 SSL 握手过程如图 1-21 所示。

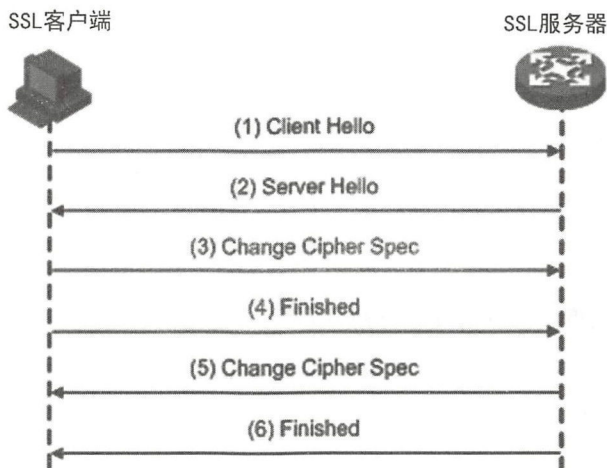


图 1-21 恢复原有会话的 SSL 握手过程

在协商会话参数、建立会话的过程中，需要使用非对称密钥算法来加密密钥、验证通信对端的身份，计算量较大，占用了大量的系统资源。为了简化 SSL 握手过程，SSL 允许重用已经协商过的会话，具体过程如下。

(1) SSL 客户端发送 Client Hello 消息，消息中的会话 ID 设置为计划重用的会话 ID。

(2) SSL 服务器如果允许重用该会话，则通过在 Server Hello 消息中设置相同的会话 ID 来应答。这样，SSL 客户端和 SSL 服务器就可以利用原有会话的密钥和加密套件，不必重新协商。

(3) SSL 客户端发送 Change Cipher Spec 消息，通知 SSL 服务器后续报文将采用原有会话的密钥和加密套件进行加密和 MAC 计算。

(4) SSL 客户端计算已交互的握手消息的 Hash 值，利用原有会话的密钥和加密套件处理 Hash 值，并通过 Finished 消息发送给 SSL 服务器，以便 SSL 服务器判断密钥和加密套件是否正确。

(5) 同样，SSL 服务器发送 Change Cipher Spec 消息，通知 SSL 客户端后续报文将采用原有会话的密钥和加密套件进行加密和 MAC 计算。

(6) SSL 服务器计算已交互的握手消息的 Hash 值，利用原有会话的密钥和加密套件处理 Hash 值，并通过 Finished 消息发送给 SSL 客户端，以便 SSL 客户端判断密钥和加密套件是否正确。

3. HTTPS

HTTPS (Hyper Text Transfer Protocol over Secure Socket Layer) 是基于 SSL 安全连接的





HTTP。HTTPS 通过 SSL 提供的数据加密、身份验证和消息完整性验证等安全机制，为 Web 访问提供安全性保证，广泛应用于网上银行、电子商务等领域。近年来，在主要互联网公司和浏览器开发者的推动之下，HTTPS 正在被加速普及，HTTP 正在被加速淘汰。不加密的 HTTP 连接是不安全的，你和目标服务器之间的任何中间人都能读取和操纵传输的数据，比如 ISP 可以在你点击的网页上插入广告，你很可能根本不知道看到的广告是否是网站发布的。中间人能够注入的代码不仅有看起来无害的广告，他们还可能注入具有恶意目的的代码。2015 年，百度联盟广告的脚本被中间人修改，加入代码对两个网站发动了 DDoS 攻击。这次攻击被称为网络大炮，网络大炮让普通的网民在不知情的状态下变成了 DDoS 攻击者，而唯一能阻止大炮的方法是加密流量。

4. 在 Tomcat 中配置 SSL/TLS 以支持 HTTPS

由于 Tomcat 是市场占有率最高的 Java 开源 Servlet 容器，下面介绍如何在 Tomcat 中配置 SSL/TLS 以支持 HTTPS。

生成密钥和证书

Tomcat 目前只能操作 JKS、PKCS11、PKCS12 格式的密钥存储库。JKS 是 Java 标准的“Java 密钥存储库”格式，是通过 keytool 命令行工具创建的。该工具包含在 JDK 中。PKCS12 格式是一种互联网标准，可以通过 OpenSSL 和 Microsoft 的 Key-Manager 来操纵。

创建一个 keystore 文件来保存服务器的私有密钥和自签名证书。

在 Windows 下执行：

```
"%JAVA_HOME%\bin\keytool" -genkey -alias tomcat -keyalg RSA
```

在 UNIX 下执行：

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA
```

执行命令后，首先会提示你提供 keystore 的密码。Tomcat 默认的密码是 changeit（全部字母都小写），当然你可以指定一个自定义密码（如果你愿意）。同样，你也需要将这个自定义密码在 server.xml 配置文件内进行指定，稍后再详述。

接下来会提示关于证书的一般信息，比如组织、联系人名称等。当用户试图在你的应用中访问一个安全页面时，该信息会显示给用户，所以一定要确保所提供的信息与用户所期望看到的内容一致。

最后，还需要输入密钥密码（key password），这个密码是这一证书（而不是存储在同一密码存储库文件中的其他证书）的专有密码。keytool 会提示，如果按下回车键，则自动使用密码存储库 keystore 的密码。当然，除了这个密码，也可以自定义密码。如果选择自定义密码，那么不要忘了在 server.xml 配置文件中指定这一密码。





下面是详细步骤：

```
C:\Users\admin>"%JAVA_HOME%\bin\keytool" -genkey -alias tomcat -keyalg RSA
```

输入密钥库口令：

再次输入新口令：

您的名字与姓氏是什么？

```
[Unknown]: waylau
```

您的单位名称是什么？

```
[Unknown]: waylau.com
```

您的组织名称是什么？

```
[Unknown]: waylau.com
```

您所在的城市或区域名称是什么？

```
[Unknown]: Hangzhou
```

您所在的省/市/自治区名称是什么？

```
[Unknown]: Zhejiang
```

该单位的双字母国家/地区代码是什么？

```
[Unknown]: china
```

```
CN=waylau, OU=waylau.com, O=waylau.com, L=hangzhou, ST=zhejiang, C=china
```

是否正确？

```
[否]: y
```

输入 <tomcat> 的密钥口令

(如果和密钥库口令相同，回车)：

如果操作全部正常，则会创建一个新的 JKS 密码存储库，该密码库包含一个自签名的证书。

该命令将在用户的主目录下创建一个新文件：.keystore。

要想指定一个不同的位置或文件名，可以在上述的 keytool 命令上添加 -keystore 参数，后面加上到达 keystore 文件的完整路径名，还需要把这个新位置指定到 server.xml 配置文件上（见后文介绍）。

Windows:

```
"%JAVA_HOME%\bin\keytool" -genkey -alias tomcat -keyalg RSA -keystore \path\to\my\keystore
```



UNIX:

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA -keystore /path/to/my/keystore
```

修改配置

取消对 Tomcat 安装目录下/conf/server.xml 中“SSL HTTP/1.1 Connector”一项的注释状态，并指定 keystore 的路径和密码：

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11NioProtocol"
    maxThreads="150" SSLEnabled="true" scheme="https" secure="true"
    keystoreFile="{user.home}/.keystore" keystorePass="changeit"
    clientAuth="false" sslProtocol="TLS" />
```

Tomcat 指定 8443 端口为 HTTPS 访问端口。如果要隐藏端口号，则要把 Tomcat 的 HTTPS 端口设为 443。

若想把所有 HTTP 请求都转到 HTTPS 上，则可以修改 Tomcat 的 conf 下的 web.xml，在 <welcome-file-list> </welcome-file-list> 节点下方添加如下代码：

```
<security-constraint>
    <!-- Authorization setting for SSL -->
    <web-resource-collection >
        <web-resource-name >SSL</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

其中，<url-pattern>是配置文件过滤策略，比如只对.jsp 的请求自动转化为 HTTPS，配置如下：

```
<security-constraint>
    <web-resource-collection >
        <web-resource-name >SSL</web-resource-name>
        <url-pattern>*.jsp</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```



在<url-pattern>中可以配置你希望自动转化的请求路径/*、login.html、login.jsp 等。

虽然 SSL 协议的意图是尽可能提供安全且高效的连接，但从性能的角度考虑，加密与解密是非常耗费计算资源的，因此将整个 Web 应用都运行在 SSL 协议下是完全没有必要的，开发者需要挑选需要安全连接的页面。对于一个相当繁忙的网站来说，通常只会在特定页面上使用 SSL 协议，也就是可能交换敏感信息的页面，比如登录页面、个人信息页面、购物车结账页面（可能会输入信用卡信息）等。应用中的任何一个页面都可以通过加密套接字来请求访问，只需将页面地址的前缀 http:换成 https:即可。

5. SSL VPN

SSL VPN 是以 SSL 为基础的 VPN 技术，利用 SSL 提供的安全机制，为用户远程访问公司内部网络提供安全保证。与复杂的 IPSec VPN 相比，SSL 通过简单易用的方法实现了信息远程连通。任何安装浏览器的机器都可以使用 SSL VPN，这是因为 SSL 内嵌在浏览器中，它不需要像传统 IPSec VPN 一样必须为每一台客户机安装客户端软件。SSL VPN 通过在远程接入用户和 SSL VPN 网关之间建立 SSL 安全连接，允许用户通过各种 Web 浏览器，各种网络接入方式，在任何地方远程访问企业网络资源，并保证企业网络的安全，保护企业内部信息不被窃取。

一般而言，SSL VPN 必须满足最基本的两个要求：

- 使用 SSL 协议进行认证和加密。没有采用 SSL 协议的 VPN 产品自然不能称为 SSL VPN，其安全性也需要进一步考证。
- 直接使用浏览器完成操作，无须安装独立的客户端。即使使用了 SSL 协议，但仍然需要分发和安装独立 VPN 客户端（如 Open VPN）的不能称为 SSL VPN，否则就失去了 SSL VPN 易于部署、免维护的优点。

SSL VPN 的典型组网模式如图 1-22 所示。

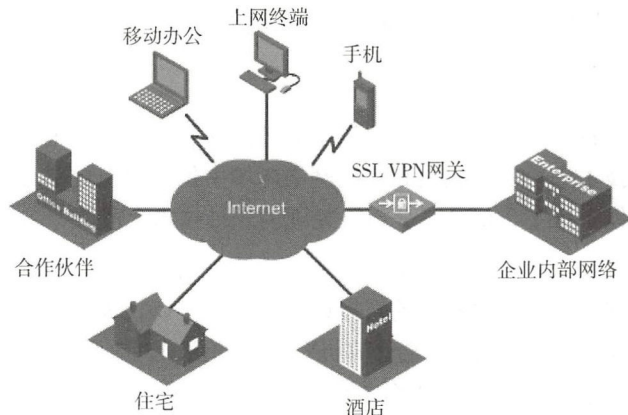


图 1-22 SSL VPN 的典型组网模式

1.7.4 访问控制

访问控制是指按用户身份及其所归属的某项定义组来限制用户对某些信息项的访问，或限制对某些控制功能的使用的一种技术。

访问控制的功能：

- 防止非法的主体进入受保护的网路资源；
- 允许合法用户访问受保护的网路资源；
- 防止合法的用户对受保护的网路资源进行非授权的访问。

1. 防火墙

防火墙指的是一个由软件和硬件设备组合而成、在内部网和外部网之间、专用网与公网之间的构造的保护屏障。这是一种获取安全性的形象说法，它是计算机硬件和软件的结合，使 Internet 与 Intranet 之间建立起一个安全网关（Security Gateway），从而保护内部网免受非法用户的侵入。防火墙主要由服务访问规则、验证工具、包过滤和应用网关 4 部分组成，防火墙就是一个位于计算机和它所连接的网络之间的软件或硬件。该计算机流入和流出的所有网络通信和数据包均要经过此防火墙。

防火墙具备以下特性：

- 内部网和外部网之间的所有网路数据流都必须经过防火墙；
- 只有符合安全策略的数据流才能通过防火墙；
- 防火墙自身应具有非常强的抗攻击免疫力；
- 应用层防火墙具备更细致的防护能力；
- 数据库防火墙具有针对数据库恶意攻击的阻断能力。

2. 堡垒机

堡垒机，即在一个特定的网路环境下，为了保障网路和数据不受来自外部和内部用户的入侵和破坏，运用各种技术手段实时收集和监控网路环境中每一个组成部分的系统状态、安全事件、网路活动，以便集中报警、记录、分析、处理的一种技术手段。

从功能上讲，它综合了核心系统运维和安全审计管控两大主干功能；从技术实现上讲，通过切断终端计算机对网路和服务器的直接访问，采用协议代理的方式，接管了终端计算机对网路和服务器的访问。形象地说，终端计算机对目标的访问，都要经过运维安全审计的翻译。打一个比方，运维安全审计扮演着看门者的角色，所有对网路设备和服务器的请求都要从这扇大门经过。因此运维安全审计能够拦截非法访问和恶意攻击，阻断不合法命令，过滤所有对目

标设备的非法访问行为，并对内部人员的误操作和非法操作进行审计监控，以便事后责任追踪。

安全审计作为企业信息安全建设不可缺少的组成部分，逐渐受到用户的关注，是企业安全体系中的重要环节。同时，安全审计是事前预防、事中预警的有效风险控制手段，也是事后追溯的可靠证据来源。

3. 拒绝服务

DoS (Denial of Service, 拒绝服务) 攻击是指通过向服务器发送大量垃圾信息或干扰信息的方式，使服务器无法向正常用户提供服务的现象。对付一个或多个资源的 DoS 往往比较容易，而要对付 DDoS (Distributed Denial of Service, 分布式拒绝服务) 则困难得多。

DDoS 攻击主要分为两类。

- **带宽耗竭的攻击**：向某个机器发送大量的消息，使正常的消息很难到达接收者。
- **资源耗竭的攻击**：使接收者把资源消耗在无用的消息上。

UDP 洪水攻击是指攻击者利用简单的 TCP/IP 服务，如 Chargen 和 Echo 来传送毫无用处的占满带宽的数据。通过伪造与某一主机的 Chargen 服务之间的一次 UDP 连接，回复地址指向开着 Echo 服务的一台主机，这样就在两台主机之间生成很多无用数据流，这些无用数据流会导致带宽的服务攻击。

SYN Flood 是当前最流行的 DoS 与 DDoS 的方式之一，这是一种利用 TCP 缺陷，发送大量伪造的 TCP 连接请求，使被攻击方资源耗尽 (CPU 满负荷或内存不足) 的攻击方式。

常见的防止 DDoS 的方式：

- **TCP 首包丢弃方案**。利用 TCP 的重传机制识别正常用户和攻击报文。当防御设备接到一个 IP 的 SYN 报文后，简单比对该 IP 是否存在于白名单中，存在则转发到后端。如果不存在于白名单中，则检查是否是该 IP 在一定时间段内的首次 SYN 报文，如果不是则检查是否重传报文，是重传则转发并加入白名单，不是则丢弃并加入黑名单。如果是首次 SYN 报文，则丢弃并等待一段时间以接收该 IP 的 SYN 重传报文，等待超时则判定为攻击报文并加入黑名单。
- **缓存**。尽量由设备的缓存直接返回结果来保护后端业务。大型的互联网企业会有庞大的 CDN 节点缓存内容。
- **重发**。可以是直接丢弃 DNS 报文导致 UDP 层面的请求重发，也可以是返回特殊响应强制要求客户端使用 TCP 重发 DNS 查询请求。
- **判断博文内容**。在一个 TCP 连接中，HTTP 报文太少和报文太多都是不正常的，过少可能是慢速连接攻击，过多可能是使用 HTTP 1.1 进行的 HTTP Flood 攻击。

4. 访问控制的模型

开发者需要在软件和设备中实现访问控制功能，常见的访问控制的模型有三种。

- **自主访问控制（Discretionary Access Control, DAC）模型：**不同的管理方式形成不同的访问控制方式。一种方式是由客体的属主对自己的客体进行管理，由属主自己决定是否将自己客体的访问权或部分访问权授予其他主体，这种控制方式是自主的，我们把它称为自主访问控制。在自主访问控制下，一个用户可以自主选择哪些用户可以共享他的文件。Linux 系统中有两种自主访问控制策略，一种是 9 位权限码（User-Group-Other），另一种是访问控制列表 ACL（Access Control List）。
- **强制访问控制（Mandatory Access Control, MAC）模型：**用于将系统中的信息分密级和类进行管理，以保证每个用户只能访问到那些被标明可以由他访问的信息的一种访问约束机制。通俗地讲，在强制访问控制下，用户（或其他主体）与文件（或其他客体）都被标记了固定的安全属性（如安全级、访问权限等），在每次访问发生时，系统检测安全属性以便确定一个用户是否有权访问该文件。其中多级安全（MultiLevel Secure, MLS）就是一种强制访问控制策略。
- **基于角色的访问控制模型（Role Based Access Control, RBAC）模型：**管理员定义一系列角色（role）并把它们赋予主体。系统进程和普通用户可能有不同的角色。设置对象为某个类型，主体具有相应的角色就可以访问它。这样就把管理员从定义每个用户的许可权限的烦琐工作中解放出来了。RBAC 有时被称为基于规则的、基于角色的访问控制（Rule-Based Role-Based Access Control, RB-RBAC）。它包含了根据主体的属性和策略定义的规则动态地赋予主体角色的机制。例如，你是一个网络中的主体，你想访问另一个网络中的对象，这个网络定义好了访问列表的路由器的另一端，路由器根据你的网络地址或协议，赋予你某个角色，决定了你是否被授权访问。

1.8 并发

计算机用户想当然地认为他们的系统在同一时间可以做多件事。比如，用户一边用浏览器下载视频文件，一边可以继续浏览器上浏览网页。可以做这样的事情软件称为并发软件（concurrent software）。

计算机实现多个程序的同时执行，主要基于以下原因：

- **资源利用率。**在某些情况下，程序必须等待其他外部的某个操作完成，才能往下继续执行，而在等待的过程中，该程序无法执行其他任何工作。因此，如果在等待的同时可以运行另外一个程序，则无疑提高了资源的利用率。
- **公平性。**不同的用户和程序对于计算机上的资源有着同等的使用权。一种高效的运行方

式是通过粗粒度的时间分片（time slicing）使这些用户和程序能共享计算机资源，而不是一个程序从头运行到尾，然后启动下一个程序。

- **便利性。**通常来说，在计算多个任务时，应该编写多个程序，每个程序执行一个任务并在必要时相互通信，这比只编写一个程序来计算所有任务更容易实现。

1.8.1 线程与并发

在早期的分时系统中，每个进程以串行化方式执行指令，并通过一组 I/O 指令与外部设备通信。每条被执行的指令都有相应的“下一条指令”，程序中的控制流就是按照指令集的规则确定的。

串行编程模型的优势是直观性和简单性，因为它模仿了人类的工作方式：每次只做一件事，做完再做其他事情。例如，早上起床后，先穿衣，然后下楼，吃早饭。在编程语言中，这些现实世界的动作可以进一步被抽象为一组粒度更细的动作。例如，喝茶的动作可以被细化为打开橱柜，挑选茶叶，将茶叶倒入杯中，查看茶壶的水是否够，不够要加水，将茶壶放在火炉上，点燃火炉，然后等水烧开，等等。在等水烧开这个过程中包含了一定程度的异步性。例如，在烧水过程中，你可以干等，也可以做其他事情，比如开始烤面包，或者看报纸（这就是另一个异步任务），同时留意水是否烧开了。但凡做事高效的人，总能在串行性和异步性之间找到合理的平衡点，程序也是如此。

线程允许在同一个进程中同时存在多个线程控制流。线程会共享进程范围内的资源，例如内存句柄和文件句柄，但每个线程都有各自的程序计数器、栈和局部变量。线程还提供了一种直观的分解模式来充分利用操作系统中的硬件并行性，而在同一个程序中的多个线程也可以被同时调度到多个 CPU 上运行。

毫无疑问，多线程编程使得程序任务并发成为可能。而并发控制主要是为了解决多个线程之间资源争夺等问题的。并发一般发生在数据聚合的地方，只要有聚合，就有争夺发生，传统解决争夺的方式采取线程锁机制，这是强行对 CPU 管理线程进行人为干预，线程唤醒成本高；新的无锁并发策略来源于异步编程、非阻塞 I/O 等编程模型。

1.8.2 并发与并行

The Practice of Programming 的作者 Rob Pike 对并发与并行做了如下描述：

并发是同一时间应对（dealing with）多件事情的能力；并行是同一时间动手做（doing）多件事情的能力。

并发（concurrency）属于问题域（problem domain），并行（parallelism）属于解决域（solution domain）。并行与并发的区别在于有无状态，并行计算适合无状态应用，而并发解决的是有状态的高性能问题；有状态要着力并发计算，无状态要着力并行计算，云计算要能做到这两种计算的自动伸缩。

1.8.3 并发带来的风险

多线程并发会带来如下的问题：

- **安全性问题。**在没有充分同步的情况下，多个线程中的操作执行顺序是不可预测的，甚至会产生奇怪的结果。线程间的通信主要是通过共享访问字段及其字段所引用的对象来实现的。这种形式的通信非常有效，但可能导致两种错误：线程干扰（thread interference）和内存一致性错误（memory consistency error）。
- **活跃度问题。**一个并行应用程序的及时执行能力被称为它的活跃度（liveness）。安全性的含义是“永远不发生糟糕的事情”，而活跃度则关注另外一个目标，即“某件正确的事情最终会发生”。当某个操作无法继续执行下去时，就会发生活跃度问题。在串行程序中，活跃度问题之一就是无意中造成的无限循环（死循环）。而在多线程程序中，常见的活跃度问题主要有死锁、饥饿和活锁。
- **性能问题。**在设计良好的并发应用程序中，线程能提升程序的性能，但无论如何，线程总是带来某种程度的运行时开销。这种开销主要发生在线程调度器临时关起活跃线程并转而运行另外一个线程的上下文切换操作（context switch）时，因为执行上下文切换，需要保存和恢复执行上下文，丢失局部性，并且 CPU 时间将更多地花在线程调度而不是线程运行上。当线程共享数据时，必须使用同步机制，这些机制往往会抑制某些编译器优化，使内存缓存区中的数据无效，并增加贡献内存总线的同步流量。所有这些因素都会带来额外的性能开销。

1. 死锁（Deadlock）

死锁是指两个或两个以上的线程永远被阻塞，一直等待对方的资源。

下面是一个用 Java 编写的死锁的例子。

Alphonse 和 Gaston 是朋友，都很有礼貌。礼貌的一个严格的规则是，当你给一个朋友鞠躬时，你必须保持鞠躬状态，直到你的朋友也给你鞠躬。不幸的是，这条规则有个缺陷，那就是如果两个朋友同一时间向对方鞠躬，就永远不会结束。在这个示例应用程序中，死锁模型是这样的：

```
public class Deadlock {  
    static class Friend {
```



```
private final String name;

public Friend(String name) {
    this.name = name;
}

public String getName() {
    return this.name;
}

public synchronized void bow(Friend bower) {
    System.out.format("%s: %s" + " has bowed to me!\n", this.name,
bower.getName());
    bower.bowBack(this);
}

public synchronized void bowBack(Friend bower) {
    System.out.format("%s: %s" + " has bowed back to me!\n", this.name,
bower.getName());
}

}

public static void main(String[] args) {
    final Friend alphonse = new Friend("Alphonse");
    final Friend gaston = new Friend("Gaston");
    new Thread(new Runnable() {
        public void run() {
            alphonse.bow(gaston);
        }
    }).start();
    new Thread(new Runnable() {
        public void run() {
            gaston.bow(alphonse);
        }
    }).start();
}
}
```

当它们尝试调用 `bowBack` 时两个线程将被阻塞。无论哪个线程，都永远不会结束，因为每个线程都在等待对方鞠躬。这就是死锁。

上面例子的代码可以在 <https://github.com/waylau/distributed-systems-technologies-and-cases-analysis> 的 `distributed-systems-java-demos` 程序的 `com.waylau.essentialjava.concurrency` 包下找到。

2. 饥饿（Starvation）

饥饿描述了一个线程由于访问太多共享资源导致不能执行程序的现象。这种情况一般出现在共享资源被某些“贪婪”线程占用而导致资源长时间不能被其他线程使用时。例如，假设一个对象提供一个同步的方法，往往需要很长时间返回，一个线程频繁调用该方法，其他线程若也需要频繁地同步访问同一个对象，则通常会被阻塞。

3. 活锁（Livelock）

一个线程常常响应另一个线程的动作，如果其他线程也常常响应该线程的动作，那么就可能出现活锁。与死锁的线程一样，程序无法进一步执行。然而，线程是不会阻塞的，它们只是会忙于应对彼此的恢复工作。现实中的例子是，两人面对面试图通过一条走廊：Alphonse 移动到他的左侧给 Gaston 让路，而 Gaston 移动到他的右侧想让 Alphonse 过去，两个人同时让路，但其实两人都挡住了对方，他们仍然彼此阻塞。

下面介绍几种解决并发问题的常用方法。

1.8.4 同步（Synchronization）

同步是避免线程干扰和内存一致性错误的常用手段。下面就用 Java 代码来演示几种问题，以及如何用同步解决问题。

1. 线程干扰

下面描述当多个线程访问共享数据时错误是如何出现的。

考虑下面的一个简单的类 `Counter`：

```
public class Counter {  
    private int c = 0;  
  
    public void increment() {  
        c++;  
    }  
  
    public void decrement() {
```

```
        c--;\n    }\n\n    public int value() {\n        return c;\n    }\n}\n}
```

其中的 `increment` 方法用来对 `c` 加 1；`decrement` 方法用来对 `c` 减 1。然而，在多个线程中都存在对某个 `Counter` 对象的引用，线程间的干扰就可能产生我们不想要的结果。

线程间的干扰出现在多个线程对同一个数据进行多个操作的时候，也就是出现了“交错”（`interleave`），意味着操作是由多个步骤构成的，此时在这多个步骤的执行上出现了叠加。

`Counter` 类对象的操作貌似不可能出现这种“交错”，因为其中的两个关于 `c` 的操作都很简单，只有一条语句。然而，即使是一条语句，也会被虚拟机翻译成多个步骤。这里我们不深究虚拟机具体将上面的操作翻译成了什么样的步骤，只需要知道即使简单的 C++ 这样的表达式也会被翻译成三个步骤：

- （1）获取 `c` 的当前值。
- （2）对其当前值加 1。
- （3）将增加后的值存储到 `c` 中。

表达式 `c--` 也会被按照同样的方式进行翻译，只不过第二步变成了减 1，而不是加 1。

假定线程 A 中调用 `increment` 方法，线程 B 中调用 `decrement` 方法，而调用时间基本相同。如果 `c` 的初始值为 0，那么这两个操作的“交错”顺序可能如下所示。

- （1）线程 A：获取 `c` 的值。
- （2）线程 B：获取 `c` 的值。
- （3）线程 A：对获取到的值加 1，其结果是 1。
- （4）线程 B：对获取到的值减 1，其结果是 -1。
- （5）线程 A：将结果存储到 `c` 中，此时 `c` 的值是 1。
- （6）线程 B：将结果存储到 `c` 中，此时 `c` 的值是 -1。

这样线程 A 计算的值就丢失了，也就是被线程 B 的值覆盖了。上面的这种“交错”只是其中的一种可能。在不同的系统环境中，有可能是 B 线程的结果丢失了，或者根本就不会出现错误。由于这种“交错”不可预测，线程间相互干扰造成的 bug 是很难定位和修改的。

2. 内存一致性错误

下面介绍通过共享内存出现的不一致的错误。

内存一致性错误发生在不同线程对同一数据产生不同的“看法”时。导致内存一致性错误的原因很复杂，超出了本书的描述范围。庆幸的是，程序员并不需要知道这些原因的细节，我们需要的是一种可以避免这种错误的方法。

避免出现内存一致性错误的关键在于理解 happens-before 关系。这种关系是一种简单的方法，能够确保一条语句对内存的写操作对于其他特定的语句都是可见的。为了理解这一点，我们可以考虑如下示例。假设定义了一个简单的 int 类型的字段并对其进行初始化：

```
int counter = 0;
```

该字段由两个线程共享：A 和 B。假定线程 A 对 counter 进行了自增操作：

```
counter++;
```

然后，线程 B 打印 counter 的值：

```
System.out.println(counter);
```

如果以上两条语句是在同一个线程中执行的，那么输出的结果自然是 1。如果这两条语句是在两个不同的线程中，那么输出的结果有可能是 0。这是因为没有保证线程 A 对 counter 的修改对线程 B 来说是可见的。除非程序员在这两条语句间建立了一定的 happens-before 关系。

我们可以采取多种方式建立 happens-before 关系，使用同步就是其中之一，这点我们将会在下面的节中看到。

到目前为止，我们已经看到了两种建立 happens-before 的方式：

- 当一条语句中调用了 Thread.start 方法，那么每一条和该语句已经建立了 happens-before 的语句都和新线程中的每一条语句有这种 happens-before 关系。引入并创建这个新线程的代码产生的结果对该线程来说都是可见的。
- 如果一个线程终止并导致另外的线程中调用 Thread.join 的语句返回，那么此时这个终止的线程中执行的所有语句都与随后的 join 语句的所有语句建立了这种 happens-before 关系。也就是说，终止了的线程中的代码效果对调用 join 方法的线程来说是可见的。

关于哪些操作可以建立 happens-before 关系，更多的信息请参阅“java.util.concurrent 包的概要说明”（<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility>）。

3. 同步方法

Java 编程语言中提供了两种基本的同步用语：同步方法（synchronized method）和同步语句（synchronized statement）。同步语句相对而言更复杂一些，我们将在下一节中进行描述，这里重点讨论同步方法。

我们只需要在声明方法的时候增加关键字 `synchronized` 即可：

```
public class SynchronizedCounter {  
    private int c = 0;  
  
    public synchronized void increment() {  
        c++;  
    }  
  
    public synchronized void decrement() {  
        c--;  
    }  
  
    public synchronized int value() {  
        return c;  
    }  
}
```

如果 `count` 是 `SynchronizedCounter` 类的实例，则设置其方法为同步方法会有两个效果：

- 首先，不可能出现对同一对象的同步方法的两个调用的“交错”。当一个线程在执行一个对象的同步方法的时候，其他所有调用该对象的同步方法的线程都会被挂起，直到第一个线程对该对象操作完毕。
- 其次，当一个同步方法退出时，会自动与该对象的同步方法的后续调用建立 `happens-before` 关系。这就确保了对该对象的修改对其他线程是可见的。

注意：构造函数不能使用 `synchronized` ——在构造函数前使用 `synchronized` 关键字将导致语义错误。同步构造函数是没有意义的。这是因为只有创建该对象的线程才能调用其构造函数。

警告：在创建多个线程共享的对象时，要特别小心对该对象的引用不能过早地“泄露”。例如，我们想要维护一个保存类的所有实例的列表 `instances`，则可能会在构造函数中这样写：

```
instances.add(this);
```

但是，其他线程可在该对象的构造完成之前就访问该对象。

同步方法是一种简单的避免线程相互干扰和内存一致性错误的策略：如果一个对象对多个线程都是可见的，那么所有对该对象的变量的读写都应该是通过同步方法完成的（一个例外就是 `final` 字段，它在对象创建完成后是不能被修改的，因此，在对象创建完毕后，可以通过非同步的方法对其进行安全的读取）。这种策略是有效的，但可能导致“活跃度问题”。这一点我们会在后面描述。

4. 内部锁和同步

同步是构建在被称为“内部锁（intrinsic lock）”或“监视锁（monitor lock）”的内部实体上的。在 API 中通常被称为“监视器（monitor）”。内部锁在两个方面扮演着重要的角色：保证对对象状态访问的排他性，建立对象可见性相关的 happens-before 关系。

每一个对象都有一个与之相关联的内部锁。按照传统的做法，当一个线程需要对一个对象的字段进行排他性访问并保持访问的一致性时，它必须在访问前先获取该对象的内部锁，然后才能访问，最后释放该内部锁。在线程获取对象的内部锁到释放对象的内部锁的这段时间，我们说该线程拥有该对象的内部锁。只要有一个线程已经拥有了一个内部锁，其他线程就不能再拥有该锁了。其他线程在试图获取该锁的时候被阻塞了。

当一个线程释放了一个内部锁，就会建立起该动作和后续获取该锁之间的 happens-before 关系。

5. 同步方法中的锁

当一个线程调用一个同步方法的时候，它就自动获得了该方法所属对象的内部锁，并在方法返回的时候释放该锁。即使由于出现了没有被捕获的异常而导致方法返回，该锁也会被释放。

我们可能会感到疑惑：当调用一个静态的同步方法的时候会怎样？静态方法是和类相关的，而不是和对象相关的。在这种情况下，线程获取的是该类的类对象的内部锁。这样对于静态字段的方法来说，这是由和类的实例的锁相区别的另外的一个锁来进行操作的。

6. 同步语句

另外一种创建同步代码的方式就是使用同步语句。和同步方法不同，使用同步语句必须指明要使用哪个对象的内部锁：

```
public void addName(String name) {  
    synchronized(this) {  
        lastName = name;  
        nameCount++;  
    }  
    nameList.add(name);  
}
```

在上面的示例中，方法 addName 需要对 lastName 和 nameCount 的修改进行同步，还要避免同步调用其他对象的方法（在同步代码段中调用其他对象的方法可能导致“活跃度”中描述的问题）。如果没有使用同步语句，那么将不得不使用一个单独、未同步的方法来完成对 nameList.add 的调用。

在改善并发性时，巧妙地使用同步语句能起到很大的帮助作用。例如，我们假定类 MsLunch

有两个实例字段，`c1` 和 `c2`，这两个变量绝不会被一起使用。所有对这两个变量的更新都需要进行同步，但没有理由阻止对 `c1` 的更新和对 `c2` 的更新出现交错——这样做会创建不必要的阻塞，进而降低并发性。此时，我们没有使用同步方法，或者使用和 `this` 相关的锁，而是创建了两个单独的对象来提供锁。

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void incl() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

采用这种方式时需要特别小心，必须绝对确保相关字段的访问交错是完全安全的。

7. 重入同步 (Reentrant Synchronization)

前面提到，线程不能获取已经被别的线程获取的锁，但是线程可以获取自身已经拥有的锁。允许一个线程能重复获得同一个锁就称为重入同步 (Reentrant Synchronization)。它是这样的一种情况：在同步代码中直接或间接地调用了还有同步代码的方法，两个同步代码段中使用的是同一个锁。如果没有重入同步，则在编写同步代码时需要额外小心，以免线程将自己阻塞。

1.8.5 原子访问 (Atomic Access)

下面介绍另外一种可以避免被其他线程干扰的做法的总体思路——原子访问。

在编程中，原子性动作就是指一次性有效完成的动作。原子性动作是不能在中间停止的：要么一次性执行完毕，要么就不执行。在动作没有执行完毕之前，是不会产生可见结果的。

通过前面的示例，我们已经发现了诸如 C++ 这样的自增表达式并不属于原子操作。即使是非常简单的表达式也包含了复杂的动作，这些动作可以被解释成许多别的动作。然而，的确存在一些原子操作：

- 对几乎所有的原生数据类型变量（除了 long 和 double）的读写及引用变量的读写都是原子的。
- 对所有声明为 volatile 的变量的读写都是原子的，包括 long 和 double 类型。

原子性动作是不会出现交错的，因此使用这些原子性动作时不用考虑线程间的干扰。然而，这并不意味着可以移除对原子操作的同步。因为内存一致性错误还是有可能出现的。使用 volatile 变量可以降低内存一致性错误的风险，因为任何对 volatile 变量的写操作都和后续对该变量的读操作建立了 happens-before 关系。这就意味着对 volatile 类型变量的修改对于别的线程来说是可见的。更重要的是，这意味着当一个线程读取一个 volatile 类型的变量时，它看到的不仅是对该变量的最后一次修改，还看到了导致这种修改的代码带来的其他影响。

使用简单的原子变量访问比通过同步代码来访问变量更高效，但是需要程序员更多细心的考虑，以避免内存一致性错误。这种额外的付出是否值得完全取决于应用程序的大小和复杂度。

1.8.6 无锁化设计提升并发能力

加锁是为了避免在并发环境下，同时访问共享资源产生的风险问题。那么，在并发环境下，是否必须加锁？答案是否定的。并非所有的并发都需要加锁。适当地降低锁的粒度，甚至采用无锁化的设计，更能提升并发能力。

比如，JDK 中的 ConcurrentHashMap，巧妙地采用了桶粒度的锁，避免了 put 和 get 中对整个 map 的锁定，尤其在 get 中，只对一个 HashEntry 做锁定操作，性能提升是显而易见的。

又比如，在程序中可以合理考虑业务数据的隔离性，实现无锁化的并发。例如，程序中预计会有两个并发任务，每个任务可以对所需要处理的数据进行分组。任务 1 去处理尾数为 0 到 4 的业务数据，任务 2 处理尾数为 5 到 9 的业务数据。那么，这两个并发任务所要处理的数据天然隔离的，也就不需要加锁。

1.8.7 缓存提升并发能力

有时为了提升整个网站的性能，我们会将经常访问的数据缓存起来，这样在下次查询时能快速找到这些数据。缓存系统往往有比传统的数据存储设备（如关系型数据库）更快的访问速度。

缓存的使用与系统的时效性有非常大的关系。当对系统时效性要求不高时，选择使用缓存

是极好的。当对系统的时效性要求比较高时，则并不适合使用缓存。

1.8.8 更细颗粒度的并发单元

线程是操作系统内核级别最小的并发单元。为了提供并发能力，某些编程语言提供了更细颗粒度的并发单元，比如纤程。相比于线程，纤程可以轻松实现百万的并发量，而且占用更少的硬件资源。

Java 语言虽然没有定义纤程，但仍有一些第三方库可选择，比如 Quasar。感兴趣的读者可以参阅 Quasar 的在线手册（<http://docs.paralleluniverse.co/quasar/>）。



第 2 章

分布式系统架构体系

2.1 基于对象的体系结构

在基于对象的分布式系统中，对象在分布式实现中起着极其关键的作用。从原理上来讲，所有的模块都可以被作为对象抽象出来，客户端将以调用对象的方式来获得服务和资源。

分布式对象之所以成为重要的范型，是因为它相对比较容易地把分布的特性隐藏在对象接口后面。此外，因为对象实际上可以是任何事务，所以它也是构建系统的强大范型。

面向对象技术于 20 世纪 80 年代开始用于开发分布式系统。同样，在达到高度分布式透明性的同时，通过远程服务器宿主独立对象的理念构成了开发新一代分布式系统的稳固的基础。在本节中，我们将看到基于对象的分布式系统的常用体系结构。

2.1.1 分布式对象

软件世界中的对象和现实世界中的对象类似，对象存储状态在字段里，通过方法来暴露其行为。方法对对象的内部状态进行操作，并作为对象与对象之间通信的主要机制。隐藏对象内部状态，通过方法进行所有的交互操作，这是面向对象编程的一个基本原则——数据封装，可以通过接口来使用方法。一个对象可能实现多个接口，给定的一个接口定义可能有多个对象为其提供实现。

把接口与实现这些接口的对象进行分隔，对于分布式系统是至关重要的。严格的隔离允许我们把接口放在一台机器上，而使对象本身驻留在另外一台机器上。这种组织通常称为分布式对象，如图 2-1 所示。

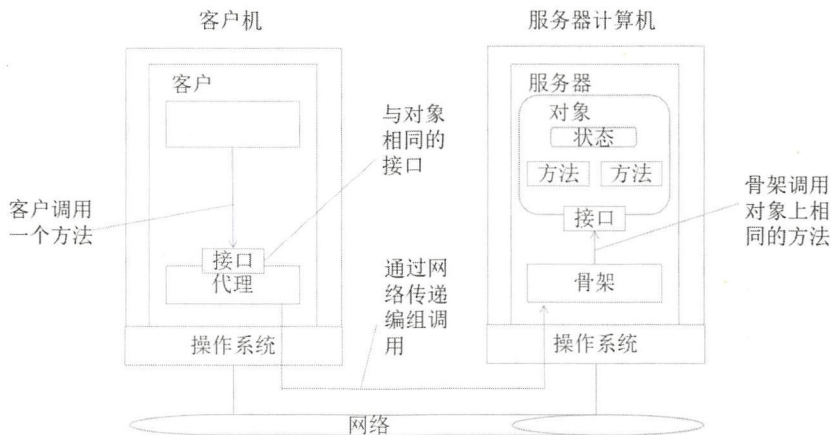


图 2-1 带有客户端代理的远程对象的常见组织

当客户绑定（bind）到一个分布式对象时，就会把这个对象的接口的实现——称为代理（proxy）——加载进客户的地址空间中。代理类似于 RPC 系统中的客户存根（client stub）。它所做的是把方法调用编组进消息中，以及对应答消息进行解组，把方法调用的结果返回给客户。实际的对象驻留在服务器计算机上，在这里提供了与它在客户机上提供的相同的接口。进入的调用请求首先被传递给服务器存根，服务器存根对它们进行解码，在服务器的对象接口上进行方法的调用。服务器存根还负责对应答进行编码，并把应答消息转发给客户端代理。

服务器端存根通常被称为骨架（skeleton），因为它提供了明确的方式，允许服务器中间件访问用户定义的对象。实际上，它通常以特定于语言的类的形式包含不完整的代码，需要开发人员进一步对其进行特殊化处理。

大多数分布式对象的一个特性是它们的状态不是分布式的。状态驻留在单台机器上，在其他机器上，智能地使用被对象实现的接口，这样的对象也被称为远程对象（remote object）。分布式对象的状态本身可能物理地分布在多台机器上，但是这种分布对于对象接口背后的客户来说是透明的。

2.1.2 Java RMI

Java 在最初只支持通过 socket 来实现分布式通信。1995 年，作为 Java 的缔造者，Sun 公司开始创建一个 Java 的扩展，称为 Java RMI（Remote Method Invocation，远程方法调用）。Java RMI 允许程序员在创建分布式应用程序时，可以从其他 Java 虚拟机（JVM）调用远程对象的方法。

一旦应用程序（客户端）引用了远程对象，就可以进行远程调用了。通过 RMI 提供的命名服务（RMI 注册中心）来查找远程对象，以接收作为返回值的引用。Java RMI 在概念上类似于 RPC，但能在不同地址空间支持对象调用的语义。

与大多数其他诸如 CORBA 的 RPC 系统不同，RMI 只支持基于 Java 来构建，但也正是这个原因，RMI 对于语言来说更加整洁，无须做额外的数据序列化工作。Java RMI 的设计目标应该是：

- 能够适应语言、集成到语言、易于使用；
- 支持无缝的远程调用对象；
- 支持服务器到 applet 的回调；
- 保障 Java 对象的安全环境；
- 支持分布式垃圾回收；
- 支持多种传输。

分布式对象模型与本地 Java 对象模型的相似点在于：

- 引用一个对象可以作为参数传递或返回的结果；
- 远程对象可以投递到任何使用 Java 语法实现的远程接口的集合上；
- 内置 Java instanceof 操作符可以用来测试远程对象是否支持远程接口。

不同点在于：

- 远程对象的类与远程接口进行交互，而不是与这些接口的实现类交互；
- Non-remote 参数对于远程方法调用来说是通过复制，而不是通过引用来传递的；
- 远程对象是通过引用来传递的，而不是复制实际的远程实现；
- 客户端必须处理额外的异常。

1. 接口和类

所有的远程接口都继承自 `java.rmi.Remote` 接口。例如：

```
public interface bankaccount extends Remote
{
    public void deposit(float amount)
        throws java.rmi.RemoteException;

    public void withdraw(float amount)
        throws OverdrawnException,
            java.rmi.RemoteException;
}
```

注意：每个方法必须在 `throws` 中声明 `java.rmi.RemoteException`。只要客户端调用远程方法出现失败，这个异常就会抛出。

2. 远程对象类

`Java.rmi.server.RemoteObject` 类提供了远程对象实现的语义，包括 `hashCode`、`equals` 和 `toString`。`java.rmi.server.RemoteServer` 及其子类让对象实现远程可见。`java.rmi.server.UnicastRemoteObject` 类定义了客户机与服务器对象实例并建立一对一的连接。

3. 存根

Java RMI 通过创建存根函数来工作。存根由 `rmic` 编译器生成。自 Java 1.5 以来，Java 支持在运行时动态生成存根类。编译器 `rmic` 会提供各种编译选项。

4. 定位对象

引导名称服务提供了用于存储对远程对象的命名引用。一个远程对象引用可以存储使用类 `java.rmi.Naming` 提供的基于 URL 的方法。例如：

```
BankAccount acct = new BankAcctImpl();
String url = "rmi://java.sun.com/account";
// 绑定 URL 到远程对象
java.rmi.Naming.bind(url, acct);

// 执行 lookup 方法
acct = (BankAccount) java.rmi.Naming.lookup(url);
```

Java RMI 的工作流程如图 2-2 所示。

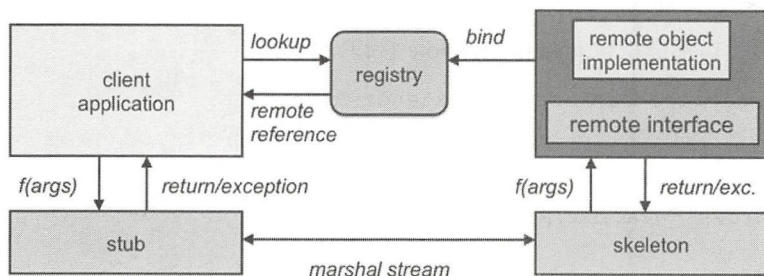


图 2-2 Java RMI 的工作流程

5. RMI 架构

RMI 是三层架构（见图 2-3），顶层是 Stub/Skeleton layer（存根/骨架层）。方法调用从 Stub、Remote Reference Layer（远程引用层）和 Transport Layer（传输层）向下传递给主机，然后再次经过 Transport Layer 层，向上穿过 Remote Reference Layer 和 Skeleton，到达服务器对象。Stub 扮演着远程服务器对象的代理的角色，使该对象可被客户激活。Remote Reference Layer 处理语义、管理单一或多重对象的通信，决定调用发往一个还是多个服务器。Transport Layer 管理实际的连接，并且追踪可以接收方法调用的远程对象。服务器端的 Skeleton 完成对服务器对象实际的方法调用，并获取返回值。返回值向下经 Remote Reference Layer、服务器端的 Transport Layer 传递回客户端，再向上经 Transport Layer 和 Remote Reference Layer 返回。最后，Stub 程序获得返回值。

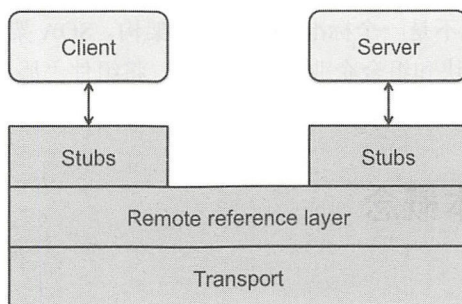


图 2-3 Java RMI 架构

要完成以上步骤需要以下几个环节：

- 生成一个远程接口；
- 实现远程对象（服务器端程序）；
- 生成 Stub 和 Skeleton（服务器端程序）；
- 编写服务器程序；
- 编写客户程序；
- 注册远程对象；
- 启动远程对象。

6. RMI 分布式垃圾回收

根据 Java 虚拟机的垃圾回收机制原理，在分布式环境下，服务器进程需要知道哪些对象不再由客户端引用，从而删除对象（垃圾回收）。在 JVM 中，Java 使用引用计数。当引用计数归零时，将进行垃圾回收。在 RMI 中，Java 支持两种操作——dirty 和 clean。本地 JVM 定期发送一个 dirty 到服务器来说明该对象仍在被使用。重发 dirty 的周期是由服务器租赁时间决定的。当客户端不需要更多的本地引用远程对象时，它发送一个 clean 调用给服务器。不像 DCOM，服务器不需要计算每个客户机使用的对象，只是简单地通知。如果它租赁时间到期之前没有接收到任何 dirty 或 clean 的消息，则可以安排将对象删除。

2.2 面向服务的架构（SOA）

SOA 体系结构（Service Oriented Architecture，面向服务的架构）基于服务组件模型，将应用程序的不同功能单元（称为服务）通过定义良好的接口契约联系起来，接口是采用中立方式进行定义的，独立于实现服务的硬件平台、操作系统和编程语言，使得构建在这样系统中的服务可以以一种统一的、通用的、灵活的方式进行交互。

SOA 不是一项技术，也不是一个标准，而是一种架构。SOA 架构独立于标准，提供了架构的蓝图。架构蓝图切开、分块和组合企业应用程序层，将组件“服务”化。SOA 中的服务与业务功能相关联，但在技术上独立于业务功能的实现。

2.2.1 SOA 的基本概念

1. SOA 的定义和组成

在 Dirk Krafzig 等人所著的 *Enterprise SOA* 一书中，对 SOA 做了如下定义：

SOA 是一个软件架构，包含四个关键概念：应用程序前端、服务、服务库和服务总线，一个服务包含一个合约、一个或多个接口及一个实现。

其中：

- 应用程序前端——业务流程的所有者；
- 服务——提供业务的功能，可以供应用程序前端或其他服务使用；
- 实现——提供业务的逻辑和数据；
- 合约——为服务客户指定功能、使用和约束；
- 接口——物理地址公开功能；
- 服务库——存储 SOA 中各个服务的服务合约；
- 服务总线——将应用程序前端和服务连在一起。

图 2-4 描述了 SOA 架构中的组成。

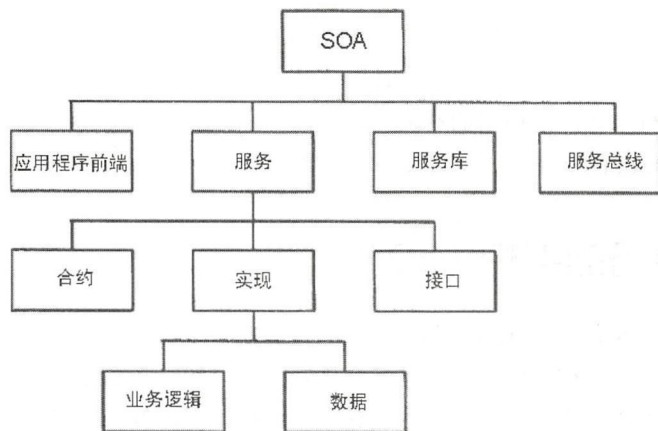


图 2-4 SOA 架构的组成

2. SOA 的角色

在 SOA 架构中，必须有如下重要实体角色，如图 2-5 所示。

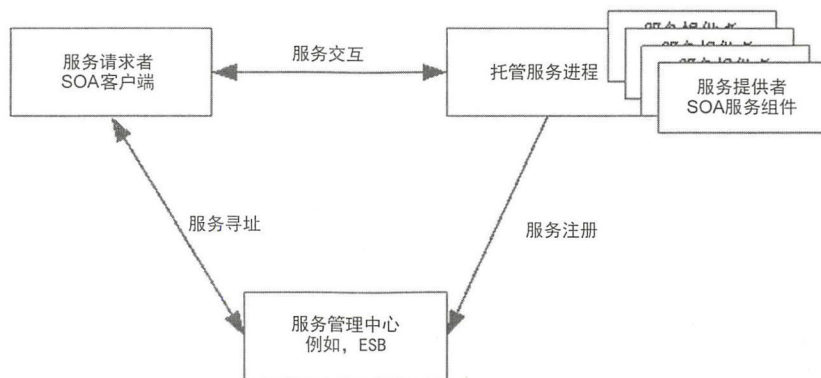


图 2-5 SOA 实体角色

- **服务请求者 (Service Customer)**：服务请求者是一个应用程序、一个软件模块或需要一个服务的另一个服务。它发起对服务管理中心中的服务查询（服务寻址），通过服务寻址，与目标服务建立通道来绑定服务，调用远程服务接口功能。服务请求者根据服务接口契约来执行远程服务。
- **服务提供者 (Service Provider)**：服务提供者是一个可通过网络寻址的进程实体（托管服务进程），与部署在托管服务进程下的 SOA 服务组件一起实现服务功能，服务提供者自动将服务组件提供的服务名发布到服务注册中心，以便服务请求者可以发现和访问该服务。服务提供者接收和执行来自请求者的请求，通过接口提供服务。
- **服务管理中心 (Service Management Center)**：服务管理中心是服务提供者与服务请求者的联系中介，提供服务提供者的服务注册管理，同时提供服务请求者的服务寻址查询。提供服务管理域中全部服务资源注册管理表，以及服务查询请求接口，允许感兴趣的服务请求者查找服务资源。

SOA 体系结构中的每个实体都扮演着服务提供者、请求者和管理中心这三种角色中的某一种（或多种）。SOA 体系结构中的操作包括：

- **服务注册**——为了使服务可访问，需要服务提供者向服务管理中心注册服务以使服务请求者可以发现和调用它。
- **服务寻址**——服务请求者定位服务，方法是查询服务注册中心来找到满足其服务需求的服务资源网络地址。
- **服务交互 (远程服务调用)**——在完成服务寻址之后，服务请求者根据与目标服务提供者建立的网络通道来调用服务。

2.2.2 基于 Web Services 的 SOA

Web Services 是 SOA 架构系统的一个实例，在 SOA 架构实现中的应用非常广泛。

由于互联网的兴起，Web 浏览器成为占主导地位的用于访问信息的模型。应用设计的首要任务大多数是让用户通过浏览器来访问，而不是通过编程访问或操作数据。

网页设计关注的是内容。解析展现方面往往是烦琐的。传统 RPC 解决方案可以工作在互联网上，但问题是，它们通常严重依赖于动态端口分配，往往要进行额外的防火墙配置。

Web Services 成为一组协议，允许服务被发布和发现，并用于技术无关的形式，即服务不应该依赖于客户的语言、操作系统或机器架构。

Web Services 的实现一般使用 Web 服务器作为服务请求的管道。客户端访问该服务，首先通过一个 HTTP 协议发送请求到服务器上的 Web 服务器。Web 服务器配置识别 URL 的一部分路径名或文件名后缀，并将请求传递给特定的浏览器插件模块。这个模块可以除去头和解析数据（如果需要），并根据需要调用其他函数或模块。对于这个实现流，一个常见的例子是浏览器对 Java Servlet 的支持。HTTP 请求会被转发到 JVM 运行的服务端代码来处理。

1. XML-RPC

XML-RPC 是在 1998 年作为一个 RPC 消息传递协议，将请求和响应封装并解析为可读的 XML 格式。XML 格式基于 HTTP 协议，避免了传统企业的防火墙需要为 RPC 服务器应用程序打开额外的端口的问题。

下面是一个 XML-RPC 消息的例子：

```
<methodCall>
  <methodName>
    sample.sumAndDifference
  </methodName>
  <params>
    <param><value><int> 5 </int></value></param>
    <param><value><int> 3 </int></value></param>
  </params>
</methodCall>
```

在这个例子中，方法 sumAndDifference 有两个整数参数（5 和 3）。

XML-RPC 支持的基本数据类型是 int、string、boolean、double 和 dateTime.iso8601。此外还有 base64 类型用于编码任意二进制数据。array 和 struct 允许定义数组和结构。

XML-RPC 不受限于任何特定的语言，也不是一套完整用于处理远程过程调用的软件，诸

如存根生成、对象管理和服务查找都不在其协议内。现在有很多库可以用于不同的语言，比如 Apache XML-RPC 可以用于 Java、Python 和 Perl。

XML-RPC 是一个简单的规范（约 7 页），没有“雄心勃勃”的目标——它只关注消息，并不处理诸如垃圾收集、远程对象、远程过程的名称服务和其他方面的问题。然而，即使没有广泛的产业支持，简单的协议却广泛被业界采用。

2. SOAP

SOAP(Simple Object Access Protocol, 简单对象访问协议)以 XML-RPC 规范作为创建 SOAP 的依据，创建于 1998 年，获得了微软和 IBM 的大力支持。该协议在创建初期只作为一种对象访问协议，但由于 SOAP 的发展，其协议已经不只用于简单的访问对象，所以这种 SOAP 缩写已经在标准的 1.2 版后被废止了。1.2 版在 2003 年 6 月 24 日成为 W3C 的推荐版本。SOAP 指定 XML 作为无状态的消息交换格式，包括 RPC 式的过程调用。

SOAP 只是一种消息格式，并未定义垃圾回收、对象引用、存根生成和传输协议。

下面是一个简单的例子：

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
      env:mustUnderstand="true">
      <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
      <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
    </m:reservation>
    <n:passenger xmlns:n="http://mycompany.example.com/employees"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
      env:mustUnderstand="true">
      <n:name>Åke Jógvan Øyvind</n:name>
    </n:passenger>
  </env:Header>
  <env:Body>
    <p:itinerary
      xmlns:p="http://travelcompany.example.org/reservation/travel">
      <p:departure>
        <p:departing>New York</p:departing>
        <p:arriving>Los Angeles</p:arriving>
      </p:departure>
    </p:itinerary>
  </env:Body>
</env:Envelope>
```



```
<p:departureDate>2001-12-14</p:departureDate>
<p:departureTime>late afternoon</p:departureTime>
<p:seatPreference>aisle</p:seatPreference>
</p:departure>
<p:return>
  <p:departing>Los Angeles</p:departing>
  <p:arriving>New York</p:arriving>
  <p:departureDate>2001-12-20</p:departureDate>
  <p:departureTime>mid-morning</p:departureTime>
  <p:seatPreference/>
</p:return>
</p:itinerary>
<q:lodging
  xmlns:q="http://travelcompany.example.org/reservation/hotels">
  <q:preference>none</q:preference>
</q:lodging>
</env:Body>
</env:Envelope>
```

其中<soap:Envelope>是 SOAP 消息中的根节点,是 SOAP 消息中必需的部分。<soap:Header>是 SOAP 消息中的可选部分,指消息头。<soap:Body>是 SOAP 中必需的部分,指消息体。

上面例子中的 SOAP 消息结构如图 2-6 所示。

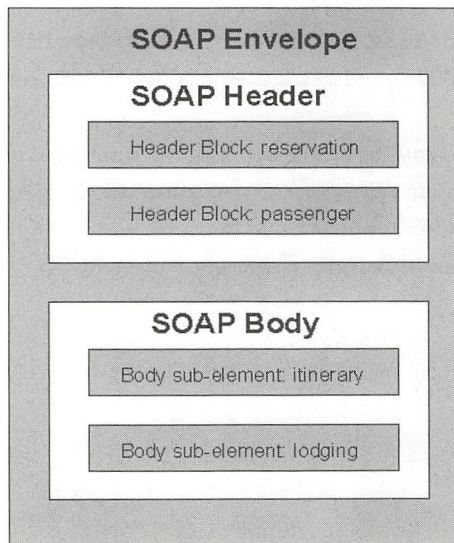


图 2-6 SOAP 消息结构





SOAP 只是提供了一个标准化的消息结构，为了实现它往往需要用 WSDL 来描述 Web Services 的方法。WSDL (Web Services Description Language) 是基于 XML 的一种用于描述 Web Services 及如何访问 Web Services 的语言。

WSDL 文档包括以下几个部分。

- 类型 (Types) : 定义 Web Services 使用的数据类型。
- 消息 (n/a) : 描述使用消息的数据元素或参数。
- 接口 (Interface) : 描述服务提供的操作，包括操作及每个操作使用的输入和输出消息。
- 绑定 (Binding) : 为每个端口定义消息格式和协议细节。例如，它可以定义 RPC 式的消息。
- 服务 (Service) : 系统功能相关的集合，包括其关联的接口、操作、消息等。
- 终点 (Endpoint) : 定义地址或 Web Services 的连接点。
- 操作 (Operation) : 定义 SOAP 的动作，以及消息编码的方式。

下面是一个 WSDL 2.0 版本的例子：

```
<?xml version="1.0" encoding="UTF-8"?>
<description xmlns="http://www.w3.org/ns/wsd1"
  xmlns:tns="http://www.tmsws.com/wsd120sample"
  xmlns:whhttp="http://schemas.xmlsoap.org/wsd1/http/"
  xmlns:wsoap="http://schemas.xmlsoap.org/wsd1/soap/"
  targetNamespace="http://www.tmsws.com/wsd120sample">

  <documentation>
    This is a sample WSDL 2.0 document.
  </documentation>

  <!-- Abstract type -->
  <types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns="http://www.tmsws.com/wsd120sample"
      targetNamespace="http://www.example.com/wsd120sample">

      <xs:element name="request"> ... </xs:element>
      <xs:element name="response"> ... </xs:element>
    </xs:schema>
```





```
</types>

<!-- Abstract interfaces -->
<interface name="Interface1">
  <fault name="Error1" element="tns:response"/>
  <operation name="Get" pattern="http://www.w3.org/ns/wsdl/in-out">
    <input messageLabel="In" element="tns:request"/>
    <output messageLabel="Out" element="tns:response"/>
  </operation>
</interface>

<!-- Concrete Binding Over HTTP -->
<binding name="HttpBinding" interface="tns:Interface1"
  type="http://www.w3.org/ns/wsdl/http">
  <operation ref="tns:Get" whttp:method="GET"/>
</binding>

<!-- Concrete Binding with SOAP-->
<binding name="SoapBinding" interface="tns:Interface1"
  type="http://www.w3.org/ns/wsdl/soap"
  wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"
  wsoap:mepDefault="http://www.w3.org/2003/05/soap/mep/request-
response">
  <operation ref="tns:Get" />
</binding>

<!-- Web Service offering endpoints for both bindings-->
<service name="Service1" interface="tns:Interface1">
  <endpoint name="HttpEndpoint"
    binding="tns:HttpBinding"
    address="http://www.example.com/rest/" />
  <endpoint name="SoapEndpoint"
    binding="tns:SoapBinding"
    address="http://www.example.com/soap/" />
</service>
</description>
```





3. Microsoft .NET Remoting

从微软的产品角度来看，可以说.NET Remoting 就是对 DCOM 的一种升级，它改善了很多功能，并极好地融合到.NET 平台下。

Microsoft .NET Remoting 提供了一种允许对象通过应用程序域与另一对象进行交互的框架。这种框架提供了多种服务，包括激活和生存期支持，以及负责与远程应用程序进行消息传输的通信通道。格式化程序用于消息在通过通道传输之前，对其进行编码和解码。应用程序可以在注重性能的场合中使用二进制编码，在需要与其他远程处理框架进行交互的场合中使用 XML 编码。从一个应用程序域向另一个应用程序域传输消息时，所有的 XML 编码都使用 SOAP 协议。出于安全性方面的考虑，远程处理提供了大量挂钩，使得在消息流通过通道进行传输之前，安全接收器能够访问消息和序列化流。

.NET Remoting 对象

有三类对象可以配置为.NET Remoting 对象，可以根据应用程序的需要选择对象类型。

- **Single Call (单一调用对象)** : Single Call 能且只能为一个请求提供服务。在需要对象完成的工作量有限的情况下 Single Call 非常有用。Single Call 对象通常不要求存储状态信息，并且不能在方法调用之间保留状态信息。但是，Single Call 对象可以配置为负载均衡模式。
- **Singleton Objects (单一元素对象)** : Singleton Objects 可以为多个客户端提供服务，因此可以通过保存客户端调用的状态信息来实现数据共享。当客户端之间需要明确地共享数据，并且不能忽略创建和维护对象的开销时，这类对象非常有用。
- **Client-Activated Objects (CAO, 客户端激活的对象)** : CAO 是服务器端的对象，根据来自客户端的请求激活这些对象。这种激活服务器对象的方法与传统的 COM coclass 激活方法非常相似。当客户端使用“new”运算符提交对服务器对象的请求时，会向远程应用程序发送一个激活请求消息。随后，服务器创建被请求类的实例，并向调用它的客户端应用程序返回 ObjRef。然后，使用此 ObjRef 在客户端上创建代理。客户端的方法调用将在代理上执行。客户端激活的对象可以为特定的客户端（不能跨越不同的客户端对象）保存方法调用之间的状态信息。每个“new”调用都会向服务器类型的独立实例返回代理。

在.NET Remoting 中，可以通过以下方式在应用程序之间传递对象：

- 作为方法调用中的参数，例如，`public int myRemoteMethod (MyRemoteObject myObj);`
- 方法调用的返回值，例如，`public MyRemoteObject myRemoteMethod(String myString);`
- 访问.NET 组件的属性或字段得到的值，例如，`myObj.myNestedObject`。





对于 Marshal By Value (MBV, 按值封送) 的对象, 当它在应用程序之间传递时, 将创建一个完整的副本。

对于 Marshal By Reference (MBR, 按引用封送) 的对象, 当它在应用程序之间传递时, 将创建该对象的引用。当对象引用 (ObjRef) 到达远程应用程序后, 将转变成“代理”返回给原始对象。

下面是一个简单.NET Remoting 服务器对象的代码示例:

```
using System;
using System.Runtime.Remoting;
namespace myRemoteService
{
    // Web Service 对象
    public class myRemoteObject : MarshalByRefObject
    {
        // myRemoteMethod 方法
        public String myRemoteMethod(String s)
        {
            return "Hello World";
        }
    }
}
```

下面是客户端代码访问这个对象的代码示例:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using myRemoteService;
public class Client
{
    public static int Main(string[] args)
    {
        ChannelServices.RegisterChannel(new HttpChannel());
        // 创建一个 myRemoteObject class 类的实例
        myRemoteObject myObj = (myRemoteObject)Activator.GetObject(typeof(
myRemoteObject),
            "http://myHost:7021/host/myRemoteObject.soap");
        myObj.myRemoteMethod ("Hello World");
    }
}
```





```
        return 0;
    }
}
```

租用生存期

对于那些在应用程序之外传送的对象，将创建一个租用。租用有一个租用时间。如果租用时间为 0，则租用过期，对象就断开与 .NET Remoting 框架的连接。一旦 AppDomain 中的所有对象引用都被释放，则在下次垃圾回收时，该对象将被回收。租用控制了对对象的生存期。

对象有默认的租用阶段。当客户端要在同一服务器对象中维护状态时，可以通过许多方法扩展租用阶段，使对象继续生存。

- 服务器对象可以将其租用时间设置为无限，这样 Remoting 在垃圾回收时就不会回收此对象。
- 客户端可以调用 RemotingServices.GetLifetimeService 方法，从 AppDomain 的租用管理器中获取服务器对象的租用。然后，客户端可以通过 Lease 对象调用 Lease.Renew 方法以延长租用。
- 客户端可将 AppDomain 的租用管理器作为特定的租用注册负责人。当 Remoting 对象的租用过期时，租用管理器将通知负责人提出续租的申请。
- 如果设置了 ILease::RenewOnCallTime 属性，则每次调用 Remoting 对象时，都会用 RenewOnCallTime 属性指定的总时间更新租用。

集成 .NET Remoting 对象

.NET Remoting 对象可以集成在：

- 托管可执行项——.NET Remoting 对象可以集成在任何常规的 .NET EXE 或托管服务中。
- .NET 组件服务——.NET Remoting 对象可以集成在 .NET 组件服务基础结构中，以便使用各种 COM+ 服务，例如，事务、JIT 和对象池等。
- IIS——.NET Remoting 对象可以集成在 Internet Information Server (IIS) 中。默认情况下，集成在 IIS 中的 Remoting 对象通过 HTTP 通道接收消息。要在 IIS 中集成 Remoting 对象，必须创建一个虚拟的根，并将 remoting.config 文件复制到 IIS 中。包含 Remoting 对象的可执行文件或 DLL 应放在 IIS 根指向的目录下的 bin 目录中。需要注意的是，IIS 根名称应该与配置文件中指定的应用程序名称相同。当第一个消息到达应用程序时，Remoting 配置文件会自动加载。使用这种方法，可以将 .NET Remoting 对象作为 Web 服务。





下面是一个 Remoting.cfg 文件的例子：

```
<configuration>
  <system.runtime.remoting>
    <application name="RemotingHello">

      <service>
        <wellknown mode="SingleCall"
          type="Hello.HelloService, Hello"
          objectUri="HelloService.soap" />
      </service>

      <channels>
        <channel port="8000"
          type="System.Runtime.Remoting.Channels.Http.HttpChannel,
            System.Runtime.Remoting" />
      </channels>

    </application>
  </system.runtime.remoting>
</configuration>
```

通道服务 (System.Runtime.Remoting.Channels)

.NET 应用程序和 AppDomain 之间使用消息进行通信。.NET 通道服务为这一通信过程提供了基础传输机制。

.NET 框架提供了 HTTP 和 TCP 通道，但是第三方也可以编写并使用自己的通道。默认情况下，HTTP 通道使用 SOAP 进行通信，TCP 通道使用二进制有效负载进行通信。

通过使用编写到集成混合应用程序中的自定义通道，可以插入通道服务（使用 IChannel）。

下面是一个加载通道服务的例子：

```
public class myRemotingObj
{
    HttpChannel httpChannel;
    TcpChannel tcpChannel;
    public void myRemotingMethod()
    {
        httpChannel = new HttpChannel();
        tcpChannel = new TcpChannel();
    }
}
```





```
// 注册 HTTP Channel
ChannelServices.RegisterChannel(httpChannel);
// 注册 TCP Channel
ChannelServices.RegisterChannel(tcpChannel);

}
}
```

序列化格式化程序 (System.Runtime.Serialization.Formatters)

.NET 序列化格式化程序对.NET 应用程序和 AppDomain 之间的消息进行编码和解码。在.NET 运行时中有两个本地格式化程序，分别为二进制 (System.Runtime.Serialization.Formatters.Binary) 和 SOAP (System.Runtime.Serialization.Formatters.Soap)。

通过实现 IRemotingFormatter 接口，并将其插入上文介绍的通道中，可以插入序列化格式化程序。这样，我们可以灵活地选择通道和格式化程序的组合方式，采用最适合应用程序的方案。

例如，可以采用 HTTP 通道和二进制格式化程序（串行化二进制数据），也可以采用 TCP 通道和 SOAP 格式。

Remoting 上下文

上下文是一个包含共享公共运行时属性的对象的范围。某些上下文属性的例子是与同步和线程紧密相关的。当.NET 对象被激活时，运行时将检查当前上下文是否一致，如果不一致，则创建新的上下文。多个对象可以同时在一个上下文中运行，并且一个 AppDomain 中可以有多个上下文。

一个上下文中的对象调用另一个上下文中的对象时，调用将通过上下文代理来执行，并且会受组合的上下文属性的强制策略影响。新对象的上下文通常是基于类的元数据属性选择的。

可以与上下文绑定的类称作上下文绑定类。上下文绑定类可以具有称为上下文属性的专用自定义属性。上下文属性是完全可扩展的，我们可以创建这些属性并将它们附加到自己的类中。与上下文绑定的对象是从 System.ContextBoundObject 导出的。

.NET Remoting 元数据和配置文件

元数据

.NET 框架使用元数据和程序集来存储有关组件的信息，使得跨语言编程成为可能。.NET Remoting 使用元数据来动态创建代理对象。在客户端创建的代理对象具有与原始类相同的成员。但是，代理对象的实现仅仅将所有请求通过.NET Remoting 运行时转发给原始对象。序列化格式化程序使用元数据在方法调用和有效负载数据流之间来回转换。





客户端可以通过以下方法获取访问 Remoting 对象所需的元数据信息。

- 服务器对象的.NET 程序集：服务器对象可以创建元数据程序集，并将其分发给客户端。在编译客户端对象时，客户端对象可以引用这些程序集。在客户端和服务端都是托管组件的封闭环境中，这种方法非常有用。
- Remoting 对象可以提供 WSDL 文件，用于说明对象及其方法。所有根据 WSDL 文件读取和生成 SOAP 请求的客户端都可以调用此对象，或使用 SOAP 与之通信。使用与 .NET SDK 一同分发的 SOAPSUDS.EXE 工具，.NET Remoting 服务器对象可以生成具有元数据功能的 WSDL 文件。当某个组织希望提供所有客户都能访问和使用的公共服务时，这种方法非常有用。
- .NET 客户可以使用 SOAPSUDS 工具从服务器上下载 XML 架构（在服务器上生成），生成仅包含元数据（没有代码）的源文件或程序集。我们可以根据需要将源文件编译到客户端应用程序中。如果多层应用程序中某一层的对象需要访问其他层的 Remoting 对象，则经常使用此方法。

配置文件

配置文件（.config 文件）用于指定给定对象的各种 Remoting 特有信息。通常情况下，每个 AppDomain 都有自己的配置文件。使用配置文件有助于实现位置的透明性。配置文件中指定的详细信息也可以通过编程来完成。使用配置文件的主要好处在于，它将与客户端代码无关的配置信息分离出来，这样在日后更改时仅需要修改配置文件，而不用编辑和重新编译源文件。.NET Remoting 的客户端和服务端对象都使用配置文件。

典型的配置文件包含以下信息及其他信息：

- 集成应用程序信息；
- 对象名称；
- 对象的 URI；
- 注册的通道（可以同时注册多个通道）；
- 服务器对象的租用时间信息。

下面是一个配置文件示例：

```
<configuration>
  <system.runtime.remoting>
    <application name="HelloNew">

      <lifetime leaseTime="20ms" sponsorshipTimeout="20ms"
renewOnCallTime="20ms" />
```





```
<client url="http://localhost:8000/RemotingHello">
  <wellknown type="Hello.HelloService, MyHello"
url="http://localhost:8000/RemotingHello/HelloService.soap" />
  <activated type="Hello.AddService, MyHello"/>
</client>

<channels>
  <channel port="8001"
type="System.Runtime.Remoting.Channels.Http.HttpChannel,
System.Runtime.Remoting" />
</channels>

</application>
</system.runtime.remoting>
</configuration>
```

.NET Remoting 方案

了解.NET Remoting 如何工作之后，让我们来看一下各种方案，分析如何在不同的方案中充分发挥.NET Remoting 的优势。表 2-1 列出了可能的客户端/服务器组合，以及默认情况下采用的底层协议和有效负载。请注意，.NET Remoting 框架是可扩展的，我们可以编写自己的通信通道和序列化格式化程序。

表 2-1 可能的客户端/服务器组合及默认情况下采用的底层协议和有效负载

客 户 端	服 务 器	有 效 负 载	协 议
.NET 组件	.NET 组件	SOAP/XML	HTTP
.NET 组件	.NET 组件	二进制	TCP
托管/非托管	.NET Web 服务	SOAP/XML	HTTP
.NET 组件	非托管的传统 COM 组件	NDR（网络数据表示形式）	DCOM
非托管的传统 COM 组件	.NET 组件	NDR	DCOM

使用 HTTP-SOAP

Web 服务是可以通过 URL 寻址的资源，并通过编程向需要使用这些资源的客户端返回信息。客户端使用 Web Services 时不必考虑其实现细节。Web Services 使用称为“合约”的严格定义的接口，此接口采用 Web 服务说明语言（WSDL）文件描述。

.NET Remoting 对象可以集成在 IIS 中，作为 Web 服务。任何可以使用 WSDL 文件的客户端都可以按照 WSDL 文件中指定的合约，对 Remoting 对象执行 SOAP 调用。IIS 使用 ISAPI 扩



展将这些请求路由到相应的对象。这样，Remoting 对象就可以作为 Web 服务对象来使用，从而充分发挥 .NET 框架基础结构的作用。如果希望不同平台/环境的程序均能够访问对象，则可以采用这种配置。这种配置便于客户端通过防火墙访问 .NET 对象。

使用 SOAP-HTTP 通道

默认情况下，HTTP 通道使用 SOAP 格式化程序，如图 2-7 所示。因此，如果客户端需要通过 Internet 访问对象，则可以使用 HTTP 通道。因为这种方法使用 HTTP 协议，所以此配置允许通过防火墙远程访问 .NET 对象。只需要按前面介绍的方法将这些对象集成在 IIS 中，即可将它配置为 Web 服务对象。随后，客户端就可以读取这些对象的 WSDL 文件，使用 SOAP 与 Remoting 对象通信。

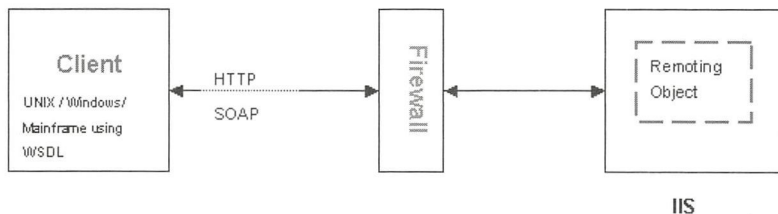


图 2-7 .NET 使用 SOAP-HTTP 通道

使用 TCP 通道

默认情况下，TCP 通道使用二进制格式化程序，如图 2-8 所示。此格式化程序以二进制格式对数据进行序列化，并使用原始 socket 在网络中传送数据。如果对象部署在受防火墙保护的封闭环境中，则此方法是理想的选择。这种方法使用 socket 在对象之间传递二进制数据，因此性能极佳。由于它使用 TCP 通道来提供对象，因此在封闭环境中具有低开销的优点。由于防火墙和配置的问题，此方法不宜在 Internet 上使用。

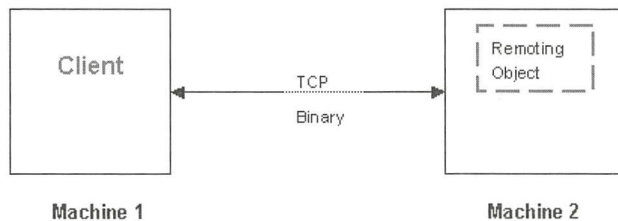


图 2-8 .NET 使用 TCP 通道

使用非托管的 COM 组件

可以通过 COM Interop Service 调用非托管的传统 COM 组件。当 .NET Remoting 客户端对象创建 COM 对象的实例时，该对象通过运行时可调用包装程序（RCW）来提供。其中，RCW 担当真正的非托管对象的代理。对于 .NET 客户，这些包装程序看起来和 .NET 客户端的任何其他

托管类一样。但实际上，它们仅仅是托管（.NET）和非托管（COM）代码之间的封送调用。

同样，我们可以将.NET Remoting 服务器对象提供给传统的 COM 客户端。当 COM 客户端创建.NET 对象的实例时，该对象通过 COM 可调用包装程序（CCW）来提供。其中，CCW 担当真正的托管对象的代理。

这两种方案都使用 DCOM 通信。如果环境中既有传统的 COM 组件，又有.NET 组件，那么这种互操作性将为我们提供便利。

总结

Microsoft .NET 框架提供了强大、可扩展、独立于语言的框架，适合开发可靠、可伸缩的分布式系统。.NET Remoting 框架提供了根据系统需求进行远程交互的强大手段。.NET Remoting 实现了与 Web 服务的无缝集成，并提供了一种方法，可以提供.NET 对象以供跨平台访问。

4. Java 中的 XML Web Services

Java RMI 与远程对象进行交互，其实现需要基于 Java 的模型。此外，它没有使用 Web Services 和基于 HTTP 的消息传递。现在已经出现了大量的软件来支持基于 Java 的 Web Services。JAX-WS（Java API for XML Web Services）就是 Web Services 消息和远程过程调用的规范。JAX-WS 的一个目标是平台互操作性，其 API 使用 SOAP 和 WSDL，双方不需要 Java 环境。

创建一个 RPC 端点

在服务器端通过下面的步骤创建一个 RPC 端点。

- 定义一个接口（Java 接口）；
- 实现服务；
- 创建一个发布者，用于创建服务的实例，并发布一个服务名字。

在客户端：

- 创建一个代理（客户端存根）——使用 `wsimport` 命令根据 WSDL 文档创建一个客户机存根；
- 编写一个客户端，通过代理创建远程服务的一个实例（存根），调用它的方法。

JAX-RPC 执行流程如下：

- Java 客户机调用存根上的方法（代理）；
- 存根调用适当的 Web 服务；
- Web 服务器被调用并执行 JAX-WS 框架；
- 框架调用实现；

- 实现返回结果给该框架；
- 该框架将结果返回给 Web 服务器；
- 服务器将结果发送给客户端存根；
- 客户端存根返回信息给调用者。

JAX-WS 调用流程如图 2-9 所示。

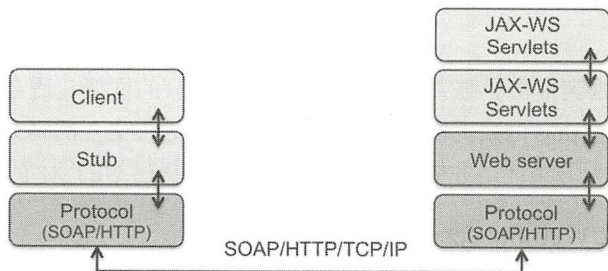


图 2-9 JAX-WS 调用流程

5. 超越 SOAP

SOAP 虽然仍然广泛用于部署应用，但在许多环境中很多厂商已经抛弃了 SOAP，转而使用其他更轻量、更容易理解，或者与 Web 交互模型更干净的机制。例如，Google 的 API 在 2006 年后就不再支持 SOAP 接口，而是使用 AJAX、XML-RPC 和 REST 作为替代者。一个匿名的微软员工批评 SOAP 过于复杂，因为“我们希望我们的工具来阅读它，而不是人”。不管上述言论是否准确，有一点是可以肯定的，SOAP 显然是一个复杂和高度冗长的格式。

AJAX

Web 浏览器最初的设计是为 Web 页面提供非动态的交互模型。Web 浏览器是建立在同步的请求-响应（request-response）上的交互模型。发送一个请求到服务器，服务器返回整个页面。当时没有更新部分页面的好方法，唯一可行的方法是利用帧，即将不同的页面加载到每一帧，其实现是笨重的，也有很大的限制性。改变了这一切的关键因素是：

- 文档对象模型（Document Object Model）和 JavaScript 的出现，使得可以以编程的方式来更改 Web 页面的各个部分；
- AJAX 提供了以非阻塞方式与服务器进行交互的方法，即允许底层 JavaScript 在等待服务器结果时，用户仍然可以与页面进行交互。

AJAX 的全称是 Asynchronous JavaScript And XML（异步的 JavaScript 和 XML），特点如下：

- 它是异步的，因为客户端在等待服务器结果时不会被阻塞。
- AJAX 集成到了 JavaScript 中，作为浏览器解释 Web 页面的一部分。JavaScript 使用

HttpRequest 来调用 AJAX 请求。JavaScript 也可能修改文档对象模型，定义页面的样子。

- 数据以 XML 文档形式发送和接收（在后期发展中，AJAX 也支持其他的数据格式，比如 JSON）。

AJAX 在推动 Web 2.0 发展的过程中发挥了重要的作用，比如产生了很多高度交互的服务，如 Google Maps、Writely 等。基本上，它允许 JavaScript 发出 HTTP 请求，获取并处理结果，刷新局部页面元素而不是整个页面。在大多数浏览器中请求的格式如下：

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.open("GET", "index.html", true);
xmlhttp.send();
```

2.2.3 SOA 的演变

随着 HTTP API、云服务、敏捷开发、持续交付、DevOps 理论的发展和实践，以及基于容器技术来部署应用和服务的日渐成熟，面向服务的架构也在不断演变，其中包括 REST 风格的架构、微服务架构、Serverless 架构等架构形式，本章的后面几节也会一一讲解这些新兴架构的风格。

2.3 REST 风格的架构

一说到 REST，很多人的第一反应就是其是前端请求后台的一种通信方式。甚至有些人将 REST 和 RPC 混为一谈，认为两者都是基于 HTTP 的类似的东西。实际上，很少人能详细讲述 REST 所提出的各个约束、风格特点，以及如何搭建 REST 服务。

本节，我们将对 REST，尤其是如何构建基于 REST 风格的 Web 服务进行详细介绍。通过本节内容，不仅可以了解什么是 REST，更能清晰地了解在编写 REST 服务时需要遵守的各个守则，识别真假 REST，设计 REST API 时需要考虑的各种因素，以及实现过程中可能遇到的问题等。

2.3.1 什么是 REST

REST (REpresentation State Transfer，表述性状态转移) 描述了一个架构样式的网络系统，比如 Web 应用程序。Roy Fielding 还是 HTTP 规范的主要编写者之一，也是 Apache HTTP 服务器项目的共同创立者。所以这篇文章一发表，就引起了极大的反响。很多公司或组织如雨后春笋般宣称自己的应用或服务实现了 REST API。但该论文实际上只是描述了一种架构风格，并未对具

体的实现进行规范。所以各大厂商不免存在误用或滥用REST。所以在这种背景下，Roy Fielding 不得不再次发文做了澄清（见*REST APIs must be hypertext-driven*¹），坦言了他的失望，并对 SocialSite REST API 提出了批评。同时他指出，除非应用状态引擎是超文本驱动的，否则它就不是REST或REST API。据此，他给出了REST API应该具备的条件：

- REST API 不应该依赖于任何通信协议，尽管要成功映射到某个协议可能会依赖于元数据的可用性、所选的方法等。
- REST API 不应该包含对通信协议的任何改动，除非是补充或确定标准协议中未规定的部分。
- REST API 应该将大部分的描述工作放在定义用于表示资源和驱动应用状态的媒体类型上，或定义现有标准媒体类型的扩展关系名和（或）支持超文本的标记。
- REST API 绝不应该定义一个固定的资源名或层次结构（客户端和服务端之间的明显耦合）。
- REST API 永远不应该有那些会影响客户端的“类型化”资源。
- REST API 不应该要求有先验知识（prior knowledge），除了初始 URI 和适合目标用户的一组标准化的媒体类型（即它能被任何潜在使用该 API 的客户端理解）。

REST 并非标准，而是一种开发 Web 应用的架构风格，可以将其理解为一种设计模式。REST 基于 HTTP、URI 及 XML 这些现有的广泛流行的协议和标准，伴随着 REST 的应用，HTTP 协议得到了更加正确的使用。

2.3.2 REST 有哪些特征

REST 指的是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是 REST。

相较于基于 SOAP 和 WSDL 的 Web 服务，REST 模式提供了更简单的实现方案。REST Web 服务（RESTful Web Services）是松耦合的，特别适用于为客户创建在互联网传播的轻量级的 Web 服务 API。REST 应用是以“资源表述的转移（the transfer of representations of resources）”为中心进行请求和响应的。数据和功能均被视为资源，并使用统一的资源标识符（URI）访问资源。网页里面的链接就是典型的 URI。该资源由文档表述，并通过使用一组简单的、定义明确的操作来执行。

例如，一个 REST 资源可能是一个城市当前的天气情况。该资源的表述可能是一个 XML

¹ <http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>。

文档、图像文件或 HTML 页面。客户端可以检索特定表述，通过更新其数据来修改资源，或者完全删除该资源。

目前，越来越多的 Web 服务开始采用 REST 风格设计和实现，真实世界中比较著名的 REST 服务包括 Google AJAX 搜索 API、Amazon Simple Storage Service (Amazon S3) 等。

基于 REST 的 Web 服务遵循一些基本的设计原则，使得 RESTful 应用更加简单、轻量，开发速度也更快：

- **通过 URI 标识资源**——系统中的每一个对象或资源都可以通过唯一的 URI 进行寻址，URI 的结构应该简单、可预测且易于理解，比如定义目录结构式的 URI。
- **统一接口**——以遵循 RFC-2616 所定义的协议的方式显式地使用 HTTP 方法，建立创建、检索、更新和删除 (CRUD: Create、Retrieve、Update 和 Delete) 操作与 HTTP 方法之间的一对一映射。
 - 若要在服务器上创建资源，则应该使用 POST 方法；
 - 若要检索某个资源，则应该使用 GET 方法；
 - 若要更新或添加资源，则应该使用 PUT 方法；
 - 若要删除某个资源，则应该使用 DELETE 方法。
- **资源多重表述**——URI 所访问的每个资源都可以使用不同的形式加以表示 (比如 XML 或 JSON)，具体的表现形式取决于访问资源的客户端，客户端与服务提供者使用一种内容协商的机制 (请求头与 MIME 类型) 来选择合适的格式，最小化彼此之间的数据耦合。在 REST 的世界中，资源即状态，而互联网就是一个巨大的状态机，每个网页是其一个状态；URI 是状态的表述；REST 风格的应用则是从一个状态迁移到下一个状态的状态转移过程。早期的互联网只有静态页面的时候，通过超链接在静态网页间浏览跳转的 `page→link→page→link...` 模式就是一种典型的状态转移过程。也就是说，早期的互联网就是天然的 REST。
- **无状态**——对服务器端的请求应该是无状态的，完整、独立的请求不要求服务器在处理请求时检索任何类型的应用程序上下文或状态。无状态约束使服务器的变化对客户端是不可见的，因为在两次连续的请求中，客户端并不依赖于同一台服务器。一个客户端从某台服务器上收到一份包含链接的文档，当它要做一些处理时，这台服务器宕机了，可能是硬盘坏掉被拿去修理，也可能是软件需要升级重启——如果这个客户端访问了从这台服务器接收的链接，则它不会察觉到后台的服务器已经改变了。通过超链接实现有状态交互，即请求消息是自包含的 (每次交互都包含完整的信息)，有多种技术实现了不同请求间状态信息的传输，例如，URI、cookies 和隐藏表单字段等，状态可以嵌入应答

消息里，这样一来状态在接下来的交互中仍然有效。REST 风格应用可以实现交互，但它却天然地具有服务器无状态的特征。在状态迁移的过程中，服务器不需要记录任何 Session，所有的状态都通过 URI 的形式记录在客户端。更准确地说，这里的无状态服务器是指服务器不保存会话状态（Session）；而资源本身则是天然的状态，通常是需要被保存的；这里的无状态服务器均指无会话状态服务器。

表 2-2 是一个 HTTP 请求方法在 RESTful Web 服务中的典型应用。

表 2-2 HTTP 请求方法在 RESTful Web 服务中的典型应用

资 源	GET	PUT	POST	DELETE
一组资源的 URI，比如 http://waylau.com/resources	列出 URI，以及该资源组中每个资源的详细信息（后者可选）	使用给定的一组资源替换当前整组资源	在本组资源中创建/追加一个新的资源。该操作往往返回新资源的 URL	删除整组资源
单个资源的 URI，比如 http://waylau.com/resources/142	获取指定的资源的详细信息，格式可以自选一个合适的网络媒体类型（比如 XML、JSON 等）	替换/创建指定的资源，并将其追加到相应的资源组中	把指定的资源当作一个资源组，并在其下创建/追加一个新的元素，使其隶属于当前资源	删除指定的元素

2.3.3 Java 实现 REST 的例子

1. Java REST 规范

针对 REST 在 Java 中的规范，主要是 JAX-RS（Java API for RESTful Web Services），该规范使得 Java 程序员可以使用一套固定的接口来开发 REST 应用，避免依赖于第三方框架。同时，JAX-RS 使用 POJO 编程模型和基于标注的配置，并集成了 JAXB，从而可以有效缩短 REST 应用的开发周期。Java EE 6 引入了对 JSR-311 的支持，Java EE 7 支持 JSR-339 规范。

JAX-RS 定义的 API 位于 javax.ws.rs 包中。

伴随着 JSR 311 规范的发布，Sun 同步发布了该规范的参考实现 Jersey。JAX-RS 的具体实现第三方还包括 Apache 的 CXF 及 JBoss 的 RESTEasy 等。未实现该规范的其他 REST 框架还包括 Spring MVC 等。

2. 基于 Jersey 的 REST 实现

在 Java 中，既然规范的制定者和实现者都是 Sun 公司（现在是 Oracle），那么毫无疑问，Jersey 就是事实上的标准，对于 Java REST 的初学者来说要尽量跟着标准走。当然，所有规范的实现在用法上基本没有差别，只是相对来说 Jersey 的实现更全面一些。

本节所有的例子都是基于 Jersey 的。若读者对 Jersey 的参考和实现感兴趣，可参阅笔者另外两本开源电子书《Jersey 2.x 用户指南》（<https://github.com/waylau/Jersey-2.x-User-Guide>）和《REST 实战》（<https://github.com/waylau/rest-in-action>）。

环境准备

- JDK 7+;
- Maven 3.2.x。

这就是所有必需的环境。当然，也可以根据自己的喜好选择使用 IDE。本书使用 Eclipse 4.4。

第一个应用

在工作目录下创建第一个 Maven 管理的应用，执行：

```
mvn archetype:generate -DarchetypeArtifactId=jersey-quickstart-webapp
-DarchetypeGroupId=org.glassfish.jersey.archetypes -DinteractiveMode=false
-DgroupId=com.waylau -DartifactId=simple-service-webapp -Dpackage=com.waylau.
rest -DarchetypeVersion=2.16
```

将项目打包成 WAR，执行：

```
mvn clean package
```

项目打包成功后（见图 2-10），可以将打包的 WAR（位于 `./target/simple-service-webapp.war`）部署到任意的 Servlet 容器，比如 Tomcat、Jetty、JBoss 等。



图 2-10 将 Maven 项目打包成 WAR

在浏览器中访问该项目，如图 2-11 所示。

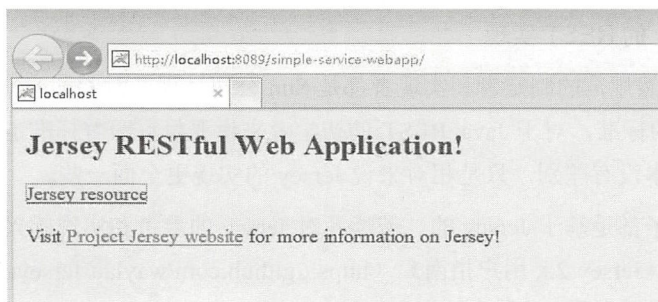


图 2-11 在浏览器中访问该项目

单击“Jersey resource”，可以在页面输出资源“Got it!”字符。

注意：部署 Jersey 项目，Servlet 容器版本应该不低于 2.5，如果想支持更高的特性（比如 JAX-RS 2.0 Async Support），则 Servlet 容器版本应该不低于 3.0。

第一个 REST 项目完成。

探索新项目

simple-service-webapp 是一个由 Jersey 提供的，Maven archetype 插件创建的 Web 项目，在你的项目里随意调整 pom.xml 内的 groupId、包号和版本号就可以创建一个新的项目。此时，simple-service-webapp 已经创建，符合 Maven 的项目结构：

- 标准的管理配置文件 pom.xml；
- 源文件路径 src/main/java；
- 资源文件路径 src/main/resources；
- Web 应用文件 src/main/webapp。

该项目包含一个名为 MyResource 的 JAX-RS 资源类。在 src/main/webapp/WEB-INF 下，它包含标准的 JavaEE Web 应用的 web.xml 部署描述符。项目中的最后一个组件是一个 index.jsp 页面，其作为这次 MyResource 资源类打包和部署的应用程序客户端。

MyResource 类是 JAX-RS 的一个实现，源代码如下：

```
package com.waylau.rest;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

/**
```



```

    * 根资源（暴露在"myresource"路径）
    */
    @Path("myresource")
    public class MyResource {

        /**
         * 方法处理 HTTP GET 请求。返回的对象以"text/plain"媒体类型
         * 给客户端
         *
         * @return String 以 text/plain 形式响应
         */
        @GET
        @Produces(MediaType.TEXT_PLAIN)
        public String getIt() {
            return "Got it!";
        }
    }
}

```

JAX-RS 资源是一个可以处理绑定了资源的 URI 的 HTTP 请求，且带有注解的 POJO。在这个例子中，单一的资源暴露了一个公开的方法，能够处理 HTTP GET 请求，绑定在/myresource URI 路径下，可以产生媒体类型为“text/plain”的响应消息。在这个示例中，资源返回相同的“Got it!”应对所有客户端的要求。

简单的 CURD 应用

下面，我们要尝试几个管理系统中常用的 CURD 操作来模拟一个“用户管理”。

服务端

在服务端，我们要提供 REST 风格的 API。

先创建一个用户对象 UserBean.java:

```

@XmlRootElement
public class UserBean {

    private int userId;
    private String name;
    private int age;

    public int getUserId() {
        return userId;
    }
}

```

```
    }

    public void setUserId(int userId) {
        this.userId = userId;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

新建一个资源类 `UserResource.java`。添加 `@Path("users")` 注解来说明资源根路径是 `users`。添加：

```
private static Map<Integer, UserBean> userMap = new HashMap<Integer, UserBean>();
```

用来在内存中存储数据。可以在 `userMap` 中获取我们想要查询的数据。

完整的代码如下：

```
@Path("users")
public class UserResource {

    private static Map<Integer, UserBean> userMap = new HashMap<Integer, UserBean>();
    /**
     * 增加
     * @param user
     */
    @POST
```

```
@Consumes(MediaType.APPLICATION_JSON)
public List<UserBean> createUser(UserBean user)
{
    userMap.put(user.getUserId(), user );
    return getAllUsers();
}

/**
 * 删除
 * @param id
 */
@DELETE
@Path("/{id}")
public List<UserBean> deleteUser(@PathParam("id")int id){
    userMap.remove(id);
    return getAllUsers();
}

/**
 * 修改
 * @param user
 */
@PUT
@Consumes(MediaType.APPLICATION_JSON)
public List<UserBean> updateUser(UserBean user){
    userMap.put(user.getUserId(), user );
    return getAllUsers();
}

/**
 * 根据 id 查询
 * @param id
 * @return
 */
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
```

```
public UserBean getUserById(@PathParam("id") int id){
    UserBean u = userMap.get(id);
    return u;
}

/**
 * 查询所有
 * @return
 */
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<UserBean> getAllUsers(){
    List<UserBean> users = new ArrayList<UserBean>();
    users.addAll( userMap.values() );
    return users;
}
}
```

简单起见，我们约定 POST 用作新增，PUT 用作修改，DELETE 用作删除，GET 用作查询。服务端接口开发完毕。

客户端

为了快速测试接口，可以用第三方 REST 客户端测试程序，这里用的是 RESTClient 插件，可以在火狐中安装使用。

我们先增加一个用户对象，使用 POST 请求发送一个 JSON 格式的数据：

```
{
    "userId": 1,
    "age": 28,
    "name": "waylau.com"
}
```

提示报错：415 未支持媒体格式的错误，如图 2-12 所示。

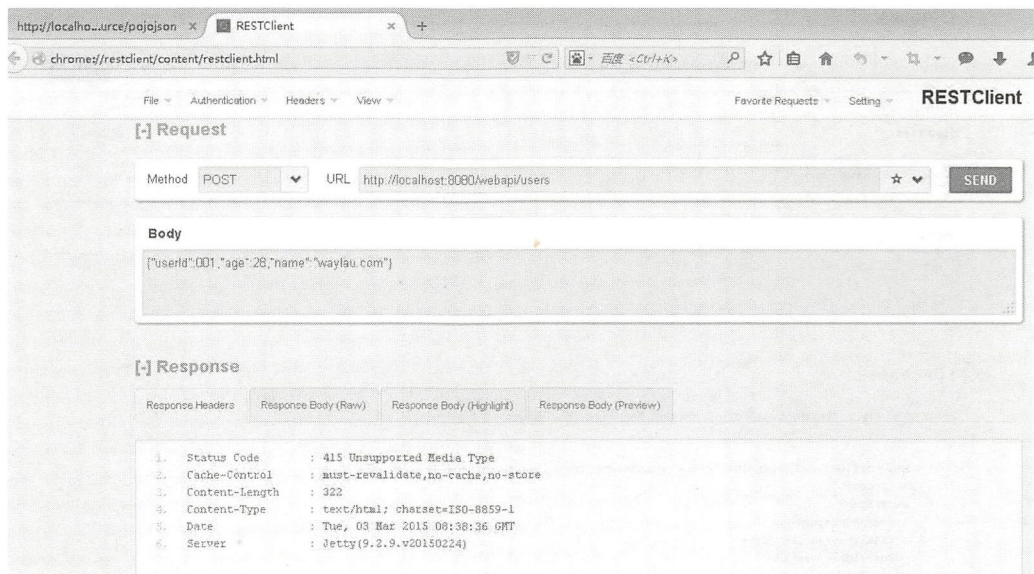


图 2-12 未支持媒体格式的错误

由于我们在新增的接口里面设置的是`@Consumes(MediaType.APPLICATION_JSON)`，规定只接收 JSON 格式，而默认的“Content-Type”是“text/html”，所以还需要在 Header 里将其设置为“application/json”，如图 2-13 所示。

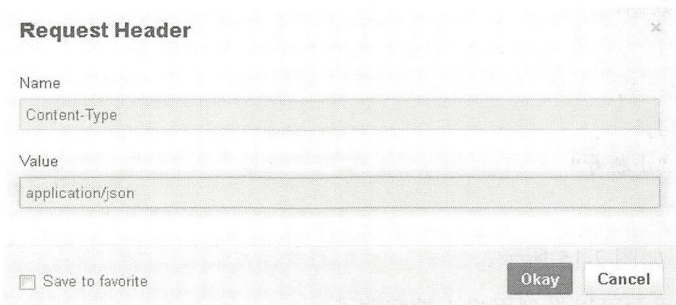


图 2-13 设置接收的媒体类型

我们再添加一个用户对象：

```
{
  "userId": 2,
  "age": 24,
  "name": "www.waylau.com"
}
```

在响应的数据里面就能看到添加的用户了，如图 2-14 所示。

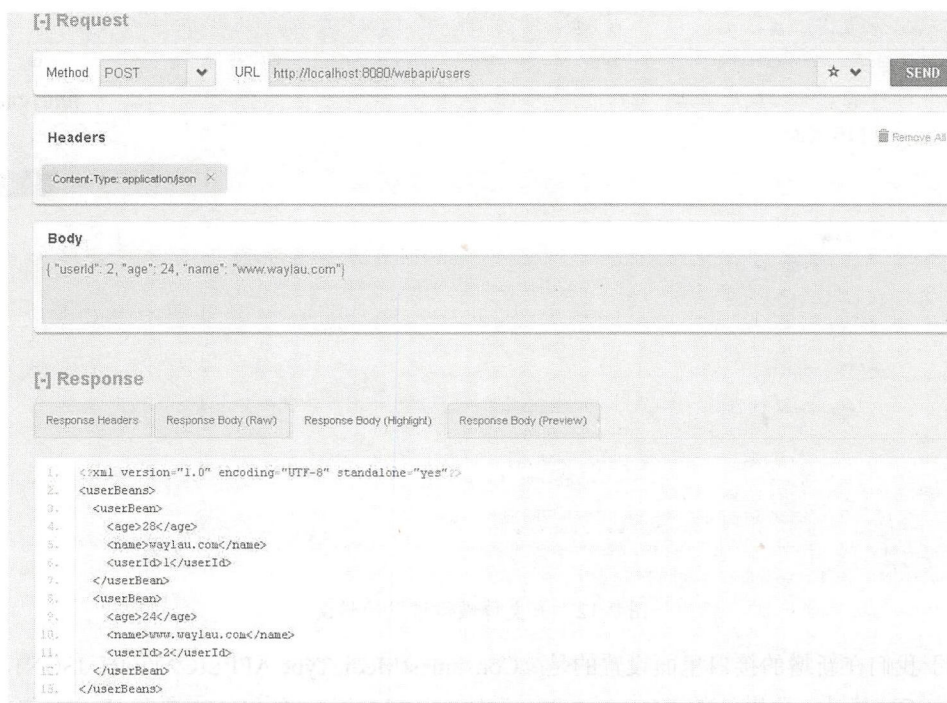


图 2-14 POST 响应

修改用户 1 的数据:

```
{
    "userId": 1,
    "age": 24,
    "name": "小柳哥"
}
```

用 PUT 请求, 如图 2-15 所示。

在返回的数据里面可以看到用户 1 被修改了。

现在模拟查询用户的操作, 在根据 ID 查询的接口里面添加如下代码:

```
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public UserBean getUserById(@PathParam("id") int id){
    UserBean u = userMap.get(id);
    return u;
}
```

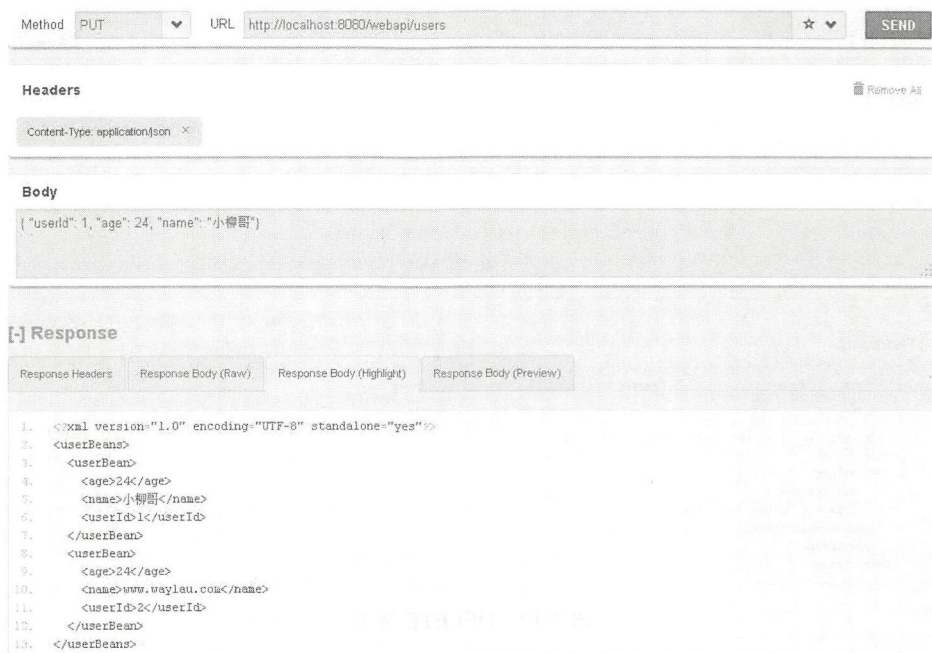


图 2-15 PUT 请求

@Path("{id}")指 id 这个子路径是一个变量。在查询用户 1 时,要将用户 1 的 userId 放在请求的 URI 里面 (http://localhost:8080/webapi/users/1), 如图 2-16 所示。

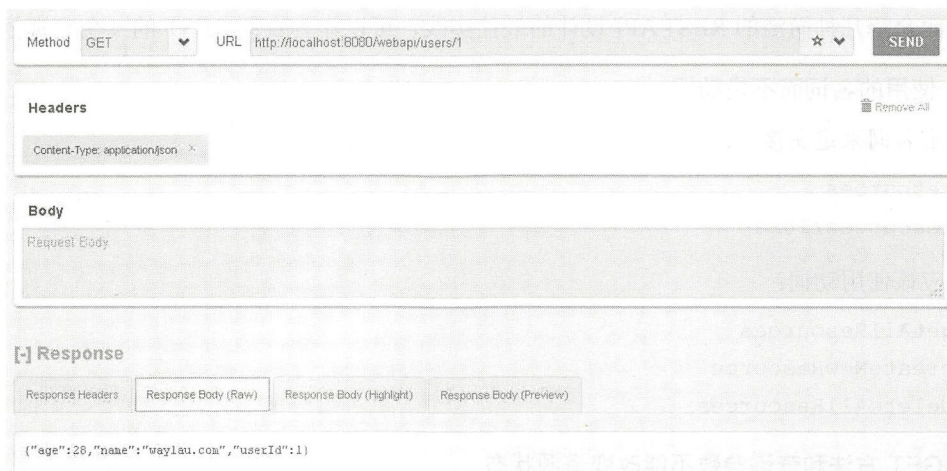


图 2-16 GET 请求

现在模拟删除用户的操作。与上面类似, 也用到了 @Path("{id}"), 如图 2-17 所示。

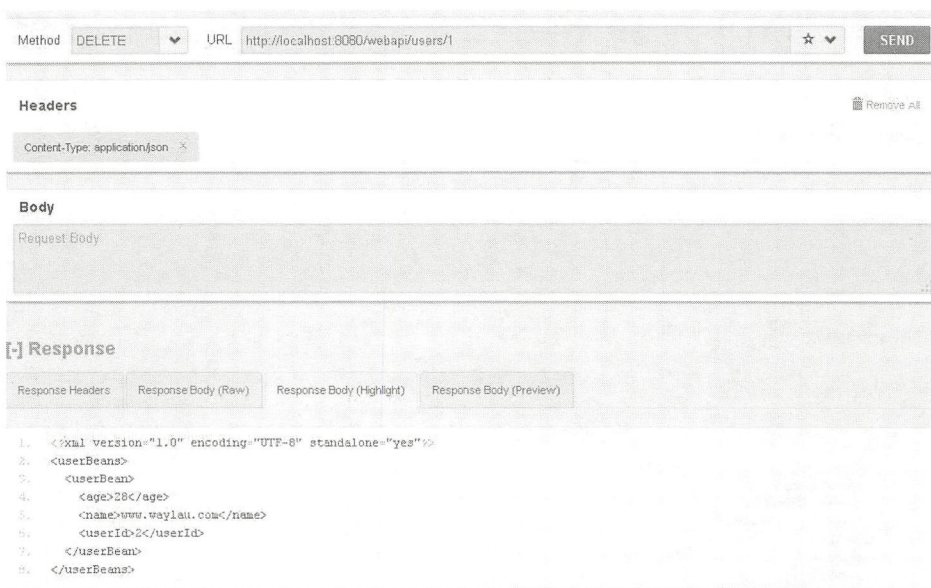


图 2-17 DELETE 请求

用户 1 被删除了。至此，整个应用完成了。这个“用户管理”够简单吧。

2.3.4 REST API 最佳实践

下面介绍几种简洁的 REST API 设计的最佳实践，可以作为真假 REST 的一个判别依据。

1. 使用的名词而不是动词

使用名词来定义接口：

```

/resources
/resources/1024

```

不应该使用动词：

```

/getAllResources
/createNewResource
/deleteAllResources

```

2. GET 方法和查询参数不能改变资源状态

如果要改变资源的状态，则要使用 PUT、POST 和 DELETE。下面使用 GET 方法修改 user 的状态是错误的：

```
GET /users/711?activate
```


或

```
GET /users/711/activate
```

3. 使用名词复数

不要混淆名词的单复数。保持简单，只用复数名词来定义所有资源。

```
/cars 代替 /car  
/users 代替 /user  
/products 代替 /product  
/settings 代替 /setting
```

4. 使用子资源来表达资源间的关系

```
GET /cars/711/drivers/ 返回 711 号 car 的所有 driver 列表  
GET /cars/711/drivers/4 返回 711 号 car 的 4 号 driver
```

5. 使用 HTTP header 来序列化格式

客户端、服务端都需要知道相互之间的通信格式。这些格式可以定义在 HTTP header 里面：

- Content-Type 定义请求格式；
- Accept 定义接收相应的格式列表。

6. 使用 HATEOAS 约束

HATEOAS(hypermedia as the engine of application state)是 REST 架构风格中最复杂的约束，也是构建成熟 REST 服务的核心。它的重要性在于打破了客户端和服务端之间严格的契约，使客户端可以更加智能和自适应，而 REST 服务本身的演化和更新也变得更加容易。在介绍 HATEOAS 之前，先介绍一下 Richardson 提出的 REST 成熟度模型。该模型把 REST 服务按照成熟度划分为 4 个层次：

- 第一个层次 (Level 0) 的 Web 服务使用 HTTP 作为传输方式，实际上只是远程方法调用 (RPC) 的一种具体形式。SOAP 和 XML-RPC 都属于此类。
- 第二个层次 (Level 1) 的 Web 服务引入了资源的概念，每个资源有对应的标识符。
- 第三个层次 (Level 2) 的 Web 服务使用不同的 HTTP 方法进行不同的操作，并且使用 HTTP 状态码表示不同的结果。例如，HTTP GET 方法用来获取资源，HTTP DELETE 方法用来删除资源。
- 第四个层次 (Level 3) 的 Web 服务使用 HATEOAS。在资源的表达中包含链接信息，客户端根据链接发现可以执行的动作。

从上述 REST 成熟度模型中可以看到，使用 HATEOAS 的 REST 服务是成熟度最高的，也是推荐的做法。对于不使用 HATEOAS 的 REST 服务，客户端和服务器的实现之间是紧密耦合的。客户端需要根据服务器提供的相关文档来了解所暴露的资源和对应的操作。当服务器发生变化，如修改了资源的 URI，客户端也需要进行相应的修改。而使用 HATEOAS 的 REST 服务中，客户端可以通过服务器提供的资源的表达来智能地发现可以执行的操作。当服务器发生变化，客户端并不需要做出修改，因为资源的 URI 和其他信息都是动态发现的。

下面是一个 HATEOAS 的例子：

```
{
  "id": 711,
  "manufacturer": "bmw",
  "model": "X5",
  "seats": 5,
  "drivers": [
    {
      "id": "23",
      "name": "Stefan Jauker",
      "links": [
        {
          "rel": "self",
          "href": "/api/v1/drivers/23"
        }
      ]
    }
  ]
}
```

7. 提供过滤、排序、字段选择、分页

过滤：

```
GET /cars?color=red
GET /cars?seats<=2
```

排序：

```
GET /cars?sort=-manufacturer,+model
```

字段选择：

```
GET /cars?fields=manufacturer,model,id,color
```

分页:

```
GET /cars?offset=10&limit=5
```

8. API 版本化

版本号使用简单的序号, 并避免使用点符号, 如 2.5 等。正确用法如下:

```
/blog/api/v1
```

9. 充分使用 HTTP 状态码来处理错误

HTTP 状态码 (HTTP Status Code) 是用来表示网页服务器 HTTP 响应状态的 3 位数字代码。它由 RFC 2616 规范定义, 并得到 RFC 2518、RFC 2817、RFC 2295、RFC 2774、RFC 4918 等规范的扩展。

在设计 API 处理错误时, 应该充分使用 HTTP 状态码, 而不是简单地抛出一个 “500 – Internal Server Error (内部服务器错误)”。

所有的异常都应该有一个错误的 payload 作为映射, 下面是一个例子:

```
{
  "errors": [
    {
      "userMessage": "Sorry, the requested resource does not exist",
      "internalMessage": "No car found in the database",
      "code": 34,
      "more info": "http://dev.mwaysolutions.com/blog/api/v1/errors/12345"
    }
  ]
}
```

2.4 微服务架构 (MSA)

自 2014 年始, 微服务 (Microservices) 一词越来越火爆, 不谈微服务仿佛就 “out” 了。那么什么是微服务? 微服务架构与传统的 SOA 架构有什么区别? 何时应该采用微服务架构? 如何构建微服务?

本节就针对上述问题, 简单介绍微服务架构。

2.4.1 什么是 MSA

微服务架构 (Microservices Architecture, MSA) 的出现并非偶然, 与这个时代的软件思想、

技术工具的发展有着密切的联系。比如，将业务功能服务化，是 SOA 的延续；RESTful 等架构的兴起，让我们可以考虑更多轻量化的通信机制；领域驱动设计指导我们如何分析并模型化复杂的业务；敏捷方法论帮助我们拥抱变化，快速反馈；持续集成和持续交付（CI/CD）促使我们构建更快、更可靠、更频繁的软件部署和交付能力；虚拟化和容器技术的发展，使我们简化了部署环境的创建和安装；DevOps 文化的流行及全栈自治团队的出现，使得小团队更加全功能化。这些都是推动微服务架构诞生和发展的重要因素。

实际上，业界对于微服务本身并没有一个严格的定义。James Lewis 和 Martin Fowler 对微服务架构做了如下定义：

简言之，微服务架构风格就像是把小的服务开发成单一应用的形式，运行在其自己的进程中，并采用轻量级的机制进行通信（一般是 HTTP 资源 API）。这些服务都是围绕业务能力来构建的，通过全自动部署工具来实现独立部署。这些服务可以使用不同的编程语言和不同的数据存储技术，并保持最小化集中管理。

MSA 包含如下特征：

- **组件以服务形式来提供**——正如其名，微服务也是面向服务的。
- **围绕业务功能进行组织**——微服务更倾向于围绕业务功能对服务结构进行划分、拆解。这样的服务是针对业务领域有着完整实现的软件，它包含使用接口、持久存储及对应的交互。因此团队应该是跨职能的，包含完整的开发技术——用户体验、数据库及项目管理。
- **产品不是项目**——传统的开发模式致力于提供一些被认为是完整的软件。一旦开发完成，软件将移交给维护或实施部门，然后开发组就可以解散了。而微服务要求开发团队对软件产品的整个生命周期负责。这要求开发者每天都关注软件产品的运行情况，并与用户紧密联系，同时承担一些售后支持。越小的服务粒度越容易促进用户与服务提供商之间的关系。Amazon 的理念就是 “You build, you run it”²，这也正是 DevOps 的文化理念。
- **强化终端及弱化通道**——微服务的应用致力于松耦合和高内聚，它们更喜欢简单的 REST 风格，而不是复杂的协议（如 WS、BPEL，或者集中式框架）。或者采用轻量级消息总线（如 RabbitMQ 或 ZeroMQ 等）来发布消息。
- **分散治理**——这是跟传统的集中式管理有很大区别的地方。微服务把整体式框架中的组件拆分成不同的服务，在构建它们时将会有更多的选择。
- **分散数据管理**——当整体式的应用使用单一逻辑数据库对数据进行持久化时，企业通常选择在应用的范围内使用一个数据库。微服务让每个服务管理自己的数据库：无论相同数据库的不同实例，还是不同的数据库系统。

² 见 <http://blog.avisi.nl/2013/09/06/you-build-it-you-run-it/>。

- **基础设施自动化**——云计算，特别是 AWS 的发展，减少了构建、发布、运维微服务的复杂性。微服务的团队更加依赖于基础设施的自动化，毕竟发布工作相当无趣。近些年火爆的容器技术，诸如 Docker 也是一个不错的选择（有关容器技术及 Docker 的内容在后面章节会涉及）。
- **容错性设计**——任何服务都可能因为供应商的不可靠而出现故障。微服务应为每个应用的服务及数据中心提供日常的故障检测和恢复。
- **改进设计**——由于设计会不断更改，微服务所提供的服务应该能够替换或报废，而不是要长久地发展。

2.4.2 MSA 与 SOA

微服务架构（MSA）与面向服务架构（SOA）有相似之处，比如，都是面向服务，通信大多基于 HTTP 协议。通常传统的 SOA 意味着大而全的单体架构（Monolithic Architecture）的解决方案。单体架构有时也被称为“单块架构”，这种架构风格给设计、开发、测试、发布都增加了难度，其中任何细小的代码变更，都将导致整个系统需要重新测试、部署。而微服务架构恰恰把所有服务都打散，设置合理的颗粒度，各个服务间保持低耦合，每个服务都在其完整生命周期中存活，将互相之间的影响降到最低。

SOA 需要对整个系统进行规范，而 MSA 的每个服务都可以有自己的开发语言、开发方式，灵活性大大提高。

1. 单体架构的例子

我们假设在构建一个电子商务应用，应用从客户处接收订单，验证库存和可用额度，并派送订单。应用包含多个组件，包括 UI 组件（用来实现用户接口），以及一些后台服务（用于检测信用额度、维护库存和派送订单）。

应用作为一体应用部署。例如，一个 Java Web 应用运行在 Tomcat 之类的 Web 容器上，仅包含单个 WAR 文件；一个 Rails 应用使用 Phusion Passenger 部署在 Apache/Nginx 上，或者使用 JRuby 部署在 Tomcat 上，它们都仅包含单个目录结构。为了伸缩和提高可用性，我们可以在一个负载均衡器下面运行该应用的多份实例。

单体架构的开发、测试、部署和扩展如图 2-18 所示。

这个方案有一些好处：

- **易于开发**——当前开发工具和 IDE 的目标就是支持这种一体应用的开发；
- **易于部署**——只需要将 WAR 文件或目录结构放到合适的运行环境下即可；
- **易于伸缩**——只需要在负载均衡器下面运行应用的多份副本就可以实现伸缩。



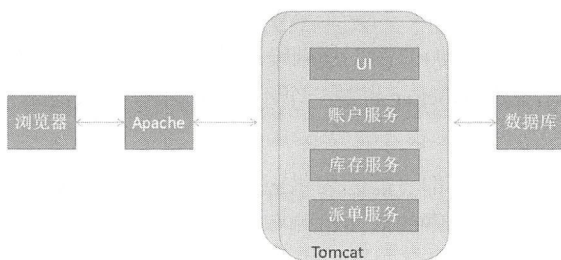


图 2-18 单体架构的开发、测试、部署和扩展

但是，一旦应用变大、团队增长，这种方法的缺点就愈加明显：

- **代码库庞大**——巨大的一体代码库可能会吓到开发者，尤其是团队的新人。应用难于理解和修改。因此，开发速度通常会减缓。另外，由于没有模块硬边界，模块化也随着时间的推移而破坏。还有，因为难于理解如何实现变更，代码质量也随着时间的推移而逐渐下降。这是个恶性循环！
- **IDE 超载**——代码库越大，IDE 越慢，开发效率越低。
- **Web 容器超载**——应用越大，容器启动时间就越长。因此开发者大量的时间被浪费在等待容器启动上。这也会影响部署。
- **难于持续部署**——对于频繁部署，巨大的单体架构应用也是个问题。为了更新一个组件，你必须重新部署整个应用。这还会中断后台任务（如 Java 应用的 Quartz 作业），不管变更是否影响这些任务，这都有可能引发问题。未被更新的组件也可能因此不能正常启动。鉴于重新部署的相关风险会增加，不鼓励频繁更新。尤其对用户界面的开发者来说，因为他们通常需要快速迭代，频繁重新部署。
- **难于伸缩应用**——单体架构只能在一个维度伸缩。一方面，它可以通过运行多个副本来伸缩以满足业务量的增加。某些云服务甚至可以动态地根据负载调整应用实例的数量。另一方面，该架构不能通过伸缩来满足数据量的增加。每个应用实例都要访问全部数据，这使得缓存低效，并且提升了内存占用和 I/O 流量。而且，不同的组件所需的资源不同，有些可能是 CPU 密集型的，另一些可能是内存密集型的。在单体架构下，我们不能独立地伸缩各个组件。
- **难于调整开发规模**——单体应用对调整开发规模也是个障碍。一旦应用达到一定规模，将工程组织分成专注于特定功能模块的团队通常更有效。比如，我们可能需要 UI 团队、会计团队、库存团队等。单体架构应用的问题是阻碍组织团队相互独立地工作。团队之间必须在开发进度和重新部署上进行协调。对团队来说也很难改变和更新产品。
- **需要对一个技术栈长期投入**——单体架构迫使你采用开发初期选择的技术栈（在某些情况下，是该项技术的某个版本）。在单体架构下，很难递增式地采用更新的技术。比如，



你选了 JVM。除了 Java 还可以选择其他使用 JVM 的语言，比如 Groovy 和 Scala 也可以与 Java 很好地进行互操作。但在单体架构下，非 JVM 语言写的组件就不行。而且，如果应用使用了后期过时的平台框架，则将应用迁移到更新更好的框架上就很有挑战性。还有可能为了采用新的平台框架，需要重写整个应用，这样就太冒险了。

微服务架构正是解决单体架构缺点的替代模式。

2. 微服务架构的例子

一个微服务架构的应用或是多层架构的，或是六角架构的，并且包含多种类型的组件：

- 表示组件（Presentation components）——响应处理 HTTP 请求，并返回 HTML 或 JSON/XML（对于 Web Service API 而言）。
- 业务逻辑（Business logic）——应用的业务逻辑。
- 数据库访问逻辑（Database access logic）——数据访问对象用于访问数据库。
- 应用集成逻辑（Application integration logic）——消息层，如基于 Spring 的集成。

这些逻辑组件分别响应应用中不同的功能模块。

最终微服务架构的解决方案如下：

- 通过采用伸缩立方（Scale Cube），特别是 y 轴方向上的伸缩来架构应用，将应用按功能分解为一组相互协作的服务的集合。每个服务实现一组有限并相关的功能。比如，一个应用可能包含订单管理服务、客户管理服务等。
- 服务间通过 HTTP/REST 等同步协议或 AMQP 等异步协议进行通信。
- 服务独立开发和部署。
- 每个服务为了与其他服务解耦，都有自己的数据库。必要时，数据库间的一致性通过数据库复制机制或应用级事件来维护。

微服务架构的服务部署如图 2-19 所示。

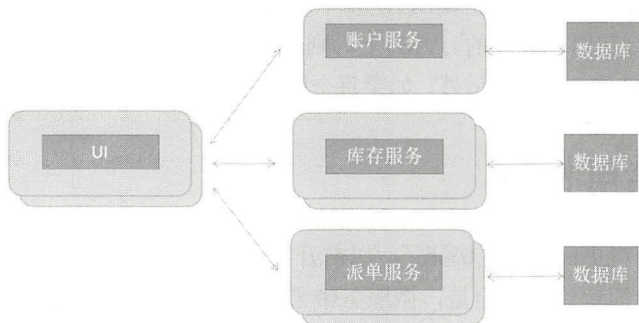


图 2-19 微服务架构的服务部署



这个方案有一些好处：

- 每个微服务都相对较小。
 - 易于开发者理解；
 - IDE 反应更快，开发更高效；
 - Web 容器启动更快，并提升了部署速度。
- 每个服务都可以独立部署，易于频繁部署新版本的服务。
- 易于伸缩开发组织结构。我们可以对多个团队的开发工作进行组织。每个团队负责单个服务。每个团队可以独立于其他团队开发、部署和伸缩服务。
- 提升故障隔离（fault isolation）。比如，如果一个服务存在内存泄漏，那么只有该服务受影响，其他服务仍然可以处理请求。相比之下，单体架构的一个出错组件可以拖垮整个系统。
- 每个服务可以单独开发和部署。
- 消除了任何对技术栈（technology stack）的长期投入。

这个方案也有一些缺点：

- 开发者要处理分布式系统的额外复杂度；
- 开发者 IDE 大多是面向构建单体架构应用的，并没有提供对开发分布式应用的支持；
- 测试更加困难；
- 开发者需要实现服务间的通信机制；
- 不使用分布式事务实现跨服务的用例更加困难；
- 实现跨服务的用例需要团队间的细致协作；
- 生产环境的部署复杂度高，对于包含多种不同服务类型的系统，部署和管理的操作复杂度仍然存在；
- 内存消耗增加。微服务架构使用 $N \times M$ 个服务实例来替代 N 个单体架构应用实例。如果每个服务运行在自己独立的 JVM 上，则通常有必要对实例进行隔离，对这么多运行的 JVM，就有 M 倍的开销。另外，如果每个服务运行在独立的虚拟机上，那么开销会更高。

2.4.3 何时采用 MSA

微服务使开发变得更简单、更快捷了。以前开发人员耗费时间来搭建环境、熟悉代码结构，在微服务的世界里会简单许多。但是，微服务带来了一系列的非功能性需求，比如说事务、服务治理（注册，发现，负载，路由，认证授权，隔离）、监控（日志，性能监控，告警，调用





链路)、部署、测试等。微服务依赖于“基础设施自动化”。

微服务不是“银弹”，何时采用微服务还需考虑企业自身的需求。

在开发应用的初期，我们通常不会遇到采用微服务这种方法来试图解决问题的情况。而且，使用这个精细、分布式的架构将会拖慢开发进度。对于初创公司，这是个严重问题，因为它们的最大挑战通常是如何快速发展业务模型及相关应用。

另一个挑战是如何将系统分隔为微服务。这是个技术活，但有些策略可能有帮助。一种方法是通过动词或用例来分隔。另一种分隔方法是通过名称或资源来分隔系统。这种服务负责对应给定类型的实体/资源的所有操作。

如果你熟悉 DDD（领域驱动设计），那么采用 DDD 来设计微服务，不但可以降低微服务环境中通用语言的复杂度，而且可以帮助团队搞清楚领域的边界，理清上下文边界。建议将每个微服务都设计成一个 DDD 限界上下文（Bounded Context）。这为系统内的微服务提供了一个逻辑边界。

理论上，每个服务应该只承担很小的职责。Bob Martin 讲过使用单一职责原则（SRP）来设计类。SRP 定义类的职责作为变化的原因，而且类应该只有一个变化的原因。使用 SRP 来设计服务也是合理的。

另一个有助于服务设计的类比是 UNIX 实用工具的设计方法。UNIX 提供了大量的实用工具如 `grep`、`cat` 和 `find`。每个工具只做一件事，通常做得非常好，并且可以跟其他工具使用 shell 脚本组合来执行复杂任务。

更多有关微服务架构设计的内容可以参阅笔者所著的《Spring Cloud 微服务架构开发实战》（<https://github.com/waylau/spring-cloud-microservices-development>）。

2.4.4 如何构建微服务

如何构建微服务真是一个大话题，本节不会涉及细节，后面章节将会详细讲解。笔者选取了自己的另外一本开源书《REST 实战》的“使用 Java SE 部署环境”一节中的例子作为实现 REST 风格 API 的微服务入门例子。该例子结合了 Jetty、Jersey 等技术。

在一体化架构中，项目经常会打包成 WAR 部署在 Tomcat 或 Jetty 等 Servlet 容器中。这种部署形式被称为基于 Servlet 的部署（Servlet-based Deployment）。

有时我们会有这样的需求，当 Web 应用不是很复杂，对应用性能要求不是很高时，需要将 HTTP Server 内嵌在 Java 程序中，只要运行 Java 程序，相应的 HTTP Server 也就跟着启动了，而且启动速度很快。这就是本节所介绍的基于 Java SE 部署环境（Java SE Deployment）来提供 REST 微服务。





1. Jetty HTTP Server

Jetty 是流行的 Servlet 容器和 HTTP 服务器。在此，我们不深究 Jetty 作为 Servlet 容器的能力（尽管在我们的测试和实例中使用它），基于 Servlet 的部署模型并没有什么特别，这里只重点描述如何使用 Jetty 的 HTTP 服务器。

Jersey 和 Jetty HTTP Server 的用法：

```
URI baseUri = UriBuilder.fromUri("http://localhost/").port(9998).build();
ResourceConfig config = new ResourceConfig(MyResource.class);
Server server = JettyHttpContainerFactory.createServer(baseUri, config);
```

要加入容器扩展模块依赖：

```
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
  <artifactId>jersey-container-jetty-http</artifactId>
  <version>2.21</version>
</dependency>
```

注意：Jetty HTTP 容器不支持部署在除了根路径是（"/"）的上下文路径中。非根路径的上下文路径在部署中是被忽略的。

2. 构建 REST 程序

构建一个实体类 MyBean.java：

```
@XmlRootElement
public class MyBean {

    private String name;
    private int age;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
```





```
        this.age = age;
    }
}
```

构建资源类 `MyResource.java`，分别向外暴露各种类型资源，包括本文、XML、JSON 格式：

```
@Path("myresource")
public class MyResource {

    /**
     * 方法处理 HTTP GET 请求。对象以"text/plain"媒体类型返回给客户端
     *
     * @return String 以 text/plain 形式响应
     */
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getIt() {
        return "Got it!";
    }

    /**
     * 方法处理 HTTP GET 请求。对象以"application/xml"媒体类型返回给客户端
     *
     * @return MyPojo 以 application/xml 形式响应
     */
    @GET
    @Path("pojoxml")
    @Produces(MediaType.APPLICATION_XML)
    public MyBean getPojoXml() {
        MyBean pojo = new MyBean();
        pojo.setName("waylau.com");
        pojo.setAge(28);
        return pojo;
    }

    /**
     * 方法处理 HTTP GET 请求。对象以"application/json"媒体类型返回给客户端
     *
     * @return MyPojo 以 application/json 形式响应
     */
}
```





```

    */
    @GET
    @Path("pojojson")
    @Produces(MediaType.APPLICATION_JSON)
    public MyBean getPojoJson() {
        MyBean pojo = new MyBean();
        pojo.setName("waylau.com");
        pojo.setAge(28);
        return pojo;
    }
}

```

程序的应用配置为 `RestApplication.java`，该配置说明了要扫描的资源包的路径 `com.waylau.rest.resource`，以及支持 JSON 转换 `MultiPartFeature`：

```

public class RestApplication extends ResourceConfig {

    public RestApplication() {
        // 资源类所在的包路径
        packages("com.waylau.rest.resource");

        // 注册 MultiPart
        register(MultiPartFeature.class);
    }
}

```

主应用程序入口类 `App.java`：

```

public class App {
    // HTTP Server 所要监听的 URI
    public static final String BASE_URI = "http://127.0.0.1:8080/";

    /**
     * Main method.
     *
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {

```





```
JettyHttpContainerFactory.createServer (URI.create (BASE_URI),  
    new RestApplication());  
}  
}
```

3. 运行程序

该程序编译后，会生成一个可执行的 `javase-rest-1.0.0.jar` 文件，用 `java -jar` 执行即可：

```
java -jar javase-rest-1.0.0.jar
```

该程序提供了如下 API：

- `http://127.0.0.1:8080/myresource;`
- `http://127.0.0.1:8080/myresource/pojoxml;`
- `http://127.0.0.1:8080/myresource/pojojson。`

上面例子的代码，可以在 <https://github.com/waylau/distributed-systems-technologies-and-cases-analysis> 的 `javase-rest` 程序中找到。

2.5 容器技术

虚拟化技术已经改变了现代计算方式，它能够提升系统资源的使用效率，消除应用程序和底层硬件之间的依赖关系，同时加强负载的可移植性和安全性，但 Hypervisor 和虚拟机只是部署虚拟负载的方式之一。作为一种能够替代传统虚拟化技术的解决方案，容器虚拟化技术凭借高效性和可靠性得到了快速发展，它能够提供新的特性，以帮助数据中心专家消除新的顾虑。

2.5.1 虚拟化技术

所谓虚拟化技术就是将事物从一种形式转变成另一种形式，最常用的虚拟化技术有操作系统中内存的虚拟化，实际运行时用户需要的内存空间可能远远大于物理机器的内存大小，利用内存的虚拟化技术，用户可以将一部分硬盘虚拟化为内存，而这对用户是透明的。又如，可以利用虚拟专用网技术（VPN）在公共网络中虚拟化一条安全、稳定的“隧道”，用户感觉像是在使用私有网络一样。

虚拟机技术是虚拟化技术的一种，虚拟机技术最早由 IBM 于 20 世纪六七十年代提出，被定义为硬件设备的软件模拟实现，通常的使用模式是分时共享昂贵的大型机。Hypervisor 是一





种运行在基础物理服务器和操作系统之间的中间软件层，可允许多个操作系统和应用共享硬件，也可称为 VMM（Virtual Machine Monitor，虚拟机监视器）。VMM 是虚拟机技术的核心，用来将硬件平台分割成多个虚拟机。VMM 运行在特权模式，主要作用是隔离并管理上层运行的多个虚拟机，仲裁它们对底层硬件的访问，并为每个客户操作系统虚拟一套独立于实际硬件的虚拟硬件环境（包括处理器、内存、I/O 设备）。VMM 采用某种调度算法在各个虚拟机之间共享 CPU，如采用时间片轮转调度算法。

2.5.2 容器与虚拟机

容器具有轻量级特性，所需的内存空间较少，提供非常快的启动速度，而虚拟机提供了专用操作系统的安全性和更牢固的逻辑边界。如果是虚拟机，那么虚拟机管理程序与硬件对话，就如同虚拟机的操作系统和应用程序构成了一个单独的物理机。虚拟机中的操作系统可以完全不同于主机的操作系统。

容器提供了更高级的隔离机制，许多应用程序在主机操作系统下运行，所有应用程序共享某些操作系统库和操作系统的内核。已经证明的屏障可以阻止运行中的容器彼此冲突，但是这种隔离存在一些安全方面的问题，我们稍后会探讨。

容器和虚拟机都具有高度可移植性，但方式不一样。就虚拟机而言，可以在运行同一虚拟机管理程序（通常是 VMware 的 ESX、微软的 Hyper-V，或者开源 Zen 和 KVM）的多个系统之间进行移植。而容器不需要虚拟机管理程序，因为它与某个版本的操作系统绑定在一起。但是容器中的应用程序可以移到任何地方，只要那里有一份该操作系统的副本即可。

容器的一大好处就是应用程序以标准方式进行格式化之后才放到容器中。开发人员可以使用同样的工具和工作流程，不管目标操作系统是什么。一旦在容器中，每种类型的应用程序都以同样的方式在网络上移动。这样一来，容器酷似虚拟机，它们又是程序包文件，可以通过互联网或内部网络来移动。

目前，应用最广泛的容器技术是 Docker。Docker 支持包括 Linux 容器、Solaris 容器、FreeBSD 容器及 Windows 容器在内的各种操作系统容器。

Docker 容器里面的应用程序无法迁移到另一个操作系统。确切地说，它能够以标准方式在网络上移动，因而更容易在数据中心内部或数据中心之间移动软件。单一容器总是与单一版本的操作系统内核关联起来的。

1. 成熟度方面的比较

虚拟机是一项高度发展、非常成熟的技术，事实证明其可以运行最关键的业务工作负载。





虚拟化软件厂商已开发出能处理成千上万个虚拟机的管理系统，那些系统旨在适合企业数据中心的现有运营。

容器代表了未来的新技术，而这种大有希望的新兴技术未必能解决每一个困难。开发人员正在开发相应的管理系统，以便一启动就将属性分配给一组容器，或者将要求相似的容器分成一组，以便组成网络或加强安全。

Docker 最初的格式化引擎正成为一种平台，并附有许多工具和 workflows。而容器获得了一些大牌技术厂商的支持。IBM、红帽、微软和 Docker 都加入了谷歌的 Kubernetes 项目，这个开源容器管理系统可用于将诸多 Linux 容器作为单一系统来管理。

Docker 有 730 家厂商为其容器平台贡献代码。CoreOS 是一款旨在运行现代基础设施堆栈的 Linux 发行版，它将广大开发人员的目光吸引到了 Rocket，这是一种新的容器运行时环境。

2. 启动速度的比较

创建容器的速度比虚拟机要快得多，那是由于虚拟机必须从存储系统检索 10GB 至 20GB 的空间给操作系统。容器中的工作负载使用主机服务器的操作系统内核，避免了这一步。容器可以实现秒级启动。

拥有这么快的速度让开发团队可以激活项目代码，以不同的方式测试代码，或者在其网站上推出额外的电子商务容量——这一切都非常快。

3. 安全方面的比较

就目前来说，虚拟机比容器有更高的安全性。容器技术并不像看上去那么可靠。以应用 Libcontainers 作为技术支持的 Docker 为例，在 Linux 系统的工作模式下，Libcontainers 可以访问五个命名空间：流程、网络、安装、主机名和共享内存。这固然很好、很强大，然而仍然有很多重要的 Linux 核心子系统不能被容器所兼容，包括所有的设备、SELinux、Cgroup，以及 `/sys` 下的所有文件系统。这意味着，如果某位用户或应用程序获取了容器内部的超级用户权限，底层操作系统理论上可以被破解。这是一件非常糟糕的事情。

现在出现了很多保护 Docker 和其他容器技术的措施。举例来说，我们可以将一个 `/sys` 文件系统设置为“只读”，或者强制某个容器进程对特定的文件系统执行“只写”操作，或者设置网络命名空间以使其只能与特定的企业内联网交流信息。但是，这些办法都不能从根本上解决问题，如此维护容器安全需要耗费大量的时间和精力。

另一项安全问题是，很多人都在发布基于容器的应用，如果未对网络上的这些应用加以识别，则很可能会下载带有木马的应用，这样就可能给我们的服务器带来严重的安全隐患。

4. 性能方面的比较

在 2014 年，IBM 研究部门发表了一篇关于容器和虚拟机环境性能比较的论文 *An Updated*



*Performance Comparison of Virtual Machines and Linux Containers*³。这篇论文使用 Docker 和 KVM 作为研究对象，阐述了 Docker 使用 NAT 或 AUFS 时的开销，并且质疑了在虚拟机上运行容器的实践方法。

论文作者在原生、容器和虚拟化环境中运行了 CPU、内存、网络 and I/O 的 Benchmark。其中，分别使用 KVM 和 Docker 作为虚拟化和容器技术的代表。Benchmark 也包含对不同环境下 Redis 和 MySQL 负载的采样。通过小数据包和多客户端的对比发现，Redis 侧重于网络栈的性能，而 MySQL 侧重于内存、网络 and 文件系统的性能。

结果显示，在每一项测试中，Docker 的性能等同于或超出 KVM 的性能。在 CPU 和内存性能方面，KVM 和 Docker 都引入了明显但可略不计的开销。但是，对于 I/O 密集型的应用，两者都需要进行调整以减少开销带来的影响。

当使用 AUFS 存储文件时，Docker 的性能会降低。而相比之下，使用卷（volume）能够获得更好的性能。卷是一种专门设计的目录，存在于一个或多个容器内。通过这种目录能够绕过联合文件系统（union file system）。这样它就没有了存储后端可能带来的开销。默认的 AUFS 后端会引起显著的 I/O 开销，特别是当有多层目录深度嵌套的时候。

Docker 的默认网络选项是 `--net=bridge`，由于 NAT 会重写数据包，也引入了性能开销。当数据包收发率变高时，这种开销会变得很明显。可以通过使用 `--net=host` 来改善网络的性能。这个选项告诉 Docker 不要为容器创建一个独立的网络栈，并允许容器拥有宿主机网络接口的完全访问权限。但是，使用这个选项时要小心。因为它允许容器内的进程像其他根进程一样使用数值较小的端口，并允许容器内的进程访问本地网络服务，如 D-bus。这使得容器内的进程可以做一些预料之外的事情，如重启宿主机。

尽管自诞生以来，KVM 的性能有了相当大的提升，但它仍然不适用于对延时敏感或高 I/O 访问率的工作负载。因为每次进行 I/O 操作，它都会增加一些开销。这个开销对于耗时较少的 I/O 操作是有意义的，但对于耗时较长的 I/O 操作是可以忽略的。

尽管在虚拟环境中运行容器是一种常见的实践方法，但是论文中建议直接在物理的 Linux 服务器上运行它们。否则，相比于直接运行在非虚拟化的 Linux 上的方法，由于虚拟机的性能开销，这种实践方法不会得到任何额外的好处。

2.5.3 基于容器的持续部署

随着 Docker 等容器技术的纷纷涌现及开源发布，软件开发行业对于现代化应用的打包及部署方式发生了巨大的变化。在没有容器等虚拟化技术的年代，程序经常需要手工部署和测试，

³ 论文见 [http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/\\$File/rc25482.pdf](http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf)。



这种工作极其烦琐且容易出错，特别是服务器数量多的时候，重复性的工作总是令人厌烦。由于开发环境、测试环境及最终的生产环境的不一致，同样的程序，有可能在不同的环境中出现不同的问题，所以经常会出现开发人员和测试人员“扯皮”的事。开发机上没有出现问题，部署到测试服务器上就出问题了。

现在就来介绍一下如何基于容器实现持续部署上面提到的种种问题。

1. 持续部署管道

持续部署管道（continuous-deployment pipeline）是指在每次代码提交时会执行的一系列步骤。管道的目的是执行一系列任务，将一个经过完整测试的功能性服务或应用部署至生产环境。唯一的手工操作就是向代码仓库执行一次签入操作，之后的所有步骤都是自动完成的。这种流程可以在一定程度上消除人为产生的错误，从而增加可靠性。并且可让机器完成它们最擅长的工作——运行重复性的过程，而不是创新性思考，从而增加了系统的吞吐量。之所以每次提交都需要通过这个管道，原因就在于“持续”这个词。如果选择延迟这一过程的执行，例如，在某个 Sprint 结束前再运行，那么整个测试与部署过程都不再是持续的了。

延迟测试及延迟部署程序到生产环境，也就延误了发现系统潜在问题的时机，最终导致的结果是修复这些问题需要投入更多的精力，毕竟在问题发生一个月后再去尝试修复，比起在问题发生一周内进行修复的成本要高得多。与之类似的是，如果在代码提交的几分钟内立即发出 bug 的通知，那么定位该 bug 所需的时间就显得微不足道了。持续部署的意义不仅在于节省维护与 bug 修复的投入，它还能够让我们更快地将新特性发布至生产环境中。开发功能并最终交付给用户使用之间的时间越短，我们就能够越快地从受益。

我们先从一个最小的子集开始，探讨一种持续部署的方案。这个最小子集能够让我们对服务进行测试、构建及部署，如图 2-20 所示。这些任务都是必不可少的。缺少了测试，我们就无法保证该服务能够正常运行；缺少了构建，就没有什么东西可部署；而缺少了部署，用户就无法从新的发布中受益。



图 2-20 简单的测试、构建、发布流程

2. 测试（Testing）

传统的软件测试方式是对源代码进行单元测试，这种方式虽然能够带来较高的代码覆盖率，但不见得一定能保证特性按照预期的方式工作，也无法保证单独的代码单元（方法、函数、类等）的行为符合设计要求。为了对特性进行验证，往往需要进行功能性测试，这种方式偏向于黑盒测试，与代码没有直接的关联。功能性测试的一个问题在于对系统存在依赖。举例来说，

Java 应用可能需要一个特定的 JDK 版本，而 Web 应用可能需要在大量的浏览器上进行测试。在不同的系统条件组合中对相同的测试集需要进行大量重复的测试。

一个令人遗憾的事实是，许多组织的测试并不充分，这无法确保一次新的发布能够在没有人工干预的情况下部署至生产环境中。即使这些测试本身是可靠的，但往往没有将这些测试在所有可能在生产环境中出现的相同条件下运行。出现这一问题的原因与我们对基础设施的管理方式有关。以人工方式对基础设施进行设置的代价是非常高的。大多数企业都需要用到多种不同的环境。比如某个环境需要运行 Ubuntu 而另一个需要运行 Red Hat，或者某个环境需要 JDK8 而另一个环境需要 JDK7。这是一种非常费时费力的途径，尤其当这些环境作为静态环境（与之相对的是通过云计算托管的“创建与销毁”途径）时更为明显。即使为了满足各种组合而设置了足够的服务器，仍然会遇到速度与灵活性的问题。举例来说，如果某个团队决定开发一个新服务，或者使用不同的技术对某个现有的服务进行重构，从请求搭建新环境直至该环境具备完整的可操作性为止也会浪费大量时间。在这段过程中，持续部署过程将陷入停顿。如果在这种环境中添加微服务，则浪费的时间将呈指数级增长。在过去，开发者通常只需关注有限的几个应用程序，而如今则需要关注成百上千个服务。毕竟，微服务的益处包括为某个用例选择最佳技术的灵活性，以及高速的发布。我们不希望等到整个系统开发完成才去部署，而是希望完成某个属于单一微服务的功能后立即进行发布。

容器技术的出现使我们可以轻松地处理各种测试问题，因为测试和最后需要部署到生产环境的将是同一个容器，里面包含的系统运行所需要的运行时与依赖都是相同的。这样开发者在测试过程中选择应用所需的组件，通过团队所用的持续部署工具构建并运行容器，让这一容器执行所需的各种测试。当代码通过全部测试之后，就可以进入下一阶段的工作了。测试所用的容器应当在注册中心（可选择私有或公有）中进行注册，以便之后重用。除了已经提到的各种益处，在测试执行结束之后，就可以销毁该容器，使服务器回到原来的状态。如此一来，就可以使用同一台服务器（或服务集群）对全部服务进行测试了。

图 2-21 中的流程已经显得有些复杂了。

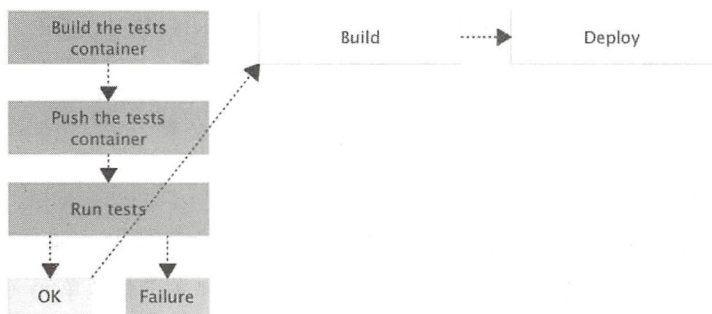


图 2-21 持续集成中的测试

3. 构建 (Building)

当执行完所有测试后, 就可以开始创建容器, 并最终将其部署至生产环境中。由于我们可能会将其部署至一个与构建所用不同的服务器中, 因此同样应当将其注册在注册中心, 如图 2-22 所示。

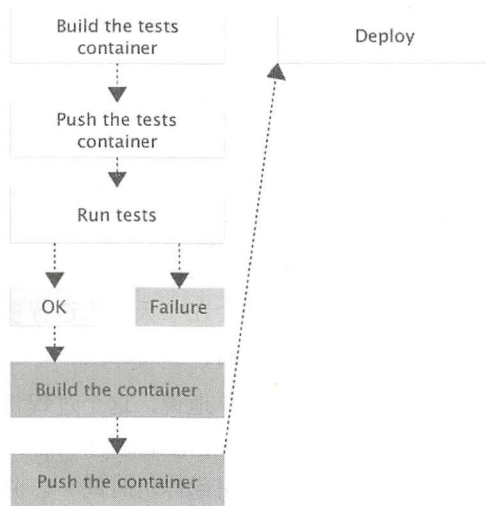


图 2-22 持续集成中的构建

当完成测试并构建好新的发布后, 就可以准备将其部署至生产服务器中。我们所要做的就是获取对应的镜像并运行容器, 如图 2-23 所示。

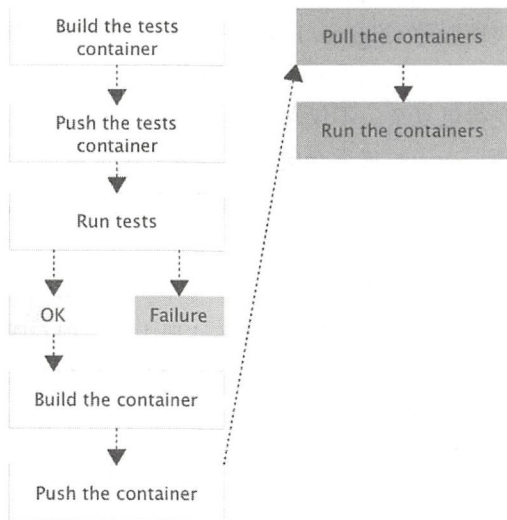


图 2-23 拉取并运行镜像

4. 部署（Deploying）

当容器上传至注册中心后，就可以在每次签入之后部署我们的微服务，并以前所未有的速度将新的特性交付给用户。

但目前所定义的流程还远远谈不上一个完整的持续部署管道。它还遗漏了许多步骤、需要考虑的内容及必需的路径。让我们依次找出这些问题并逐个解决。

5. 蓝—绿部署（Blue-Green Deployment）

整个管道中最危险的步骤可能就是部署了。如果我们获取了某个新的发布并开始运行，则容器就会以新的发布取代旧的发布。也就是说，在过程中会出现一定程度的停机时间。容器需要停止旧的发布并启动新的发布，同时我们的服务也需要进行初始化。虽然这一过程可能只需几分钟、几秒钟甚至几微秒，但还是造成了停机时间。如果实施了微服务与持续部署实践，那么发布的次数会比之前更频繁。最终，我们可能会在一天之内进行多次部署。无论决定采用怎样的发布频率，对用户的干扰都是我们应当避免的。

应对这一问题的解决方案是蓝—绿部署。简单地说，这个过程将部署一个新发布，使其与旧发布并行运行。可将某个版本称为“蓝”，另一个版本称为“绿”。由于两者是并行运行的，因此不会产生停机时间（至少不会由于部署流程引起停机时间）。并行运行两个版本的方式为我们带来了一些新的可能性，但也造成了一些新的挑战。

在实践蓝—绿部署时要考虑的第一件事就是如何将用户的请求从旧的发布重定向至新的发布。在此之前的部署方式中，我们只是简单地将旧的发布替换为新的发布，因此它们将在相同的服务器与端口上运行。而蓝—绿部署将并行运行两个版本，每个版本将使用自己的端口。有可能我们已经使用了某些代理服务（Nginx、HAProxy 等），那么我们可能会面对一个新的挑战，即这些代理不能是静态的了。在每次新发布中，代理的配置需要进行持续变更。如果在集群中进行部署，那么该过程将变得更复杂。不仅端口需要变更，IP 地址也需要变更。为了有效地使用集群，我们需要将服务部署在当时最适合的服务器上。决定最适合服务器的条件包括可用的内存、磁盘和 CPU 的类型等。通过这种方式，我们能够以最佳的方式分布服务，并极大地优化可用资源的利用率。而这又造成了新的问题，最紧迫的问题是如何找到所部署的服务的 IP 地址与端口号。对这个问题的答案是使用服务发现。

服务发现包括三个部分。首先需要通过一个服务注册中心以保存服务的信息。其次需要某个进程对新的服务进行注册，并撤销已中止的服务。最后需要通过某种方式获取服务的信息。举例来说，当部署一个新的发布时，注册进程需要在服务注册中心保存 IP 地址与端口信息。随后，代理可发现这些信息，并通过信息对本身进行重新配置。常见的服务注册中心包括 etcd、Consul 和 ZooKeeper。可以使用 Registrator 来注册和撤销服务，以及用 confd 和 Consul Template 来实现服务发现与创建模板。

现在，我们已经找到了一种保存及获取服务信息的机制，可利用该机制对代理进行重新配置，唯一一个还未解答的问题就是要部署哪个版本（颜色）。当我们进行手工部署时，自然知道之前部署的是哪个颜色。如果之前部署了绿色，那么现在当然要部署蓝色。如果一切都是自动化执行的，则需要将这一信息保存起来，让部署流程能够访问它。由于我们在流程中已经建立了服务发现功能，所以可以将部署颜色与服务 IP 地址和端口信息一同保存起来，以便在必要时获取该信息。

在完成了以上工作后，管道将变为图 2-24 中所显示的状态。由于步骤的数量增加了，因此将这些步骤划分为预部署、部署及部署后三个组。

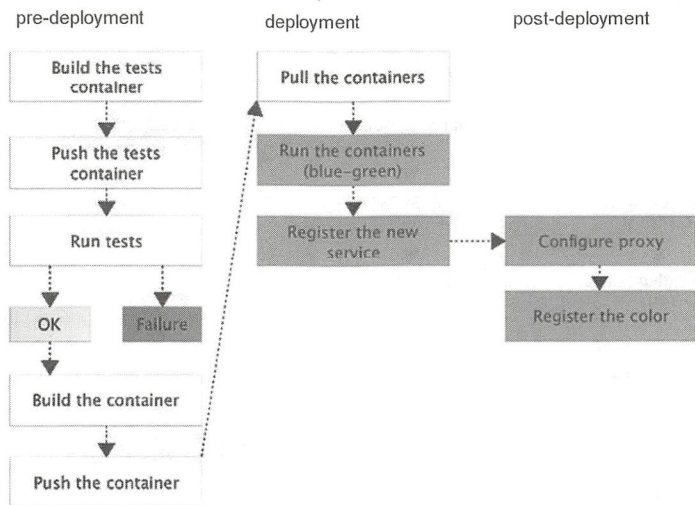


图 2-24 蓝—绿部署

6. 运行预集成以及集成后测试

虽然测试的运行至关重要，但它无法验证要部署至生产环境中的服务是否真的能够按预期运行。服务在生产环境上无法正常工作的原因是多种多样的，许多环节都有可能产生错误，可能是没有正确地安装数据库或防火墙阻碍了对服务的访问。即使代码按预期工作，也不代表已验证了部署的服务得到了正确的配置。即便搭建了一个预发布服务器以部署我们的服务，并且进行了又一轮测试，也无法完全确信在生产环境中总是能够得到相同的结果。为了区分不同类型的测试，Viktor Farcic 将其称为“预部署（pre-deployment）”测试，这些测试的相同点是在构建与部署服务之前运行。

蓝—绿流程为我们展现了一种新的机会。由于旧发布与新发布是并行运行的，我们可以对新发布进行测试，随后再对代理进行重新配置，以指向新的发布。通过这种方式，我们可以放

心地将新发布部署至生产环境并进行测试，而代理仍会将我们的用户重定向至旧的发布。Viktor Farcic 将这一阶段的测试称为“预集成（pre-integration）”测试。它意味着我们在将新发布与代理服务集成之前（在代理进行重新配置之前）所需运行的测试。这些测试可以让我们忽略预发布环境（这种环境与生产环境永远做不到完全一致），对代理重新配置之后用户将使用完全相同的配置对新发布进行测试。

最后，当我们重新配置代理之后，还需要再进行一轮测试。这一轮测试称为“集成后（post-integration）”测试，如图 2-25 所示。这一过程应当能够快速完成，因为唯一需要验证的就是代理是否确实正确配置了。通常来说，只需对 80（HTTP）与 443（HTTPS）端口进行几次请求作为测试就足够了。

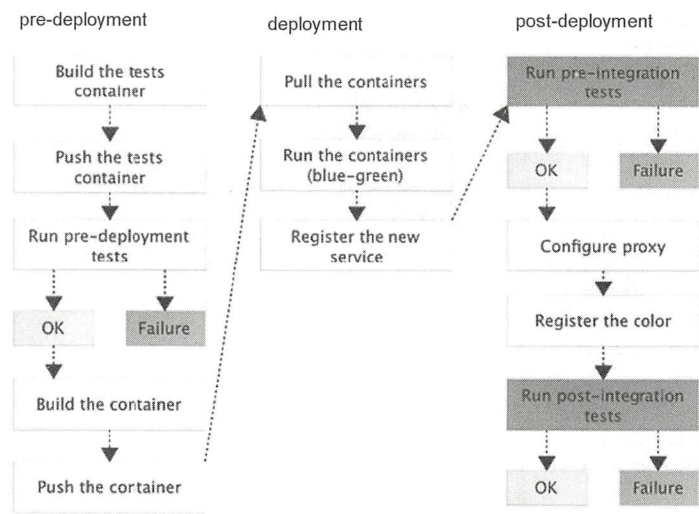


图 2-25 预集成以及集成后测试

7. 回滚与清理

如果在整个流程中有任何一部分出错，则整个环境就应当保持与该流程尚未初始化之前相同的状态，即状态回滚（rolling back）。

即使整个过程如计划般一样顺利执行，也仍然有一些清理工作需要处理。我们需要停止旧的发布，并删除其注册信息。

回滚与清理如图 2-26 所示。

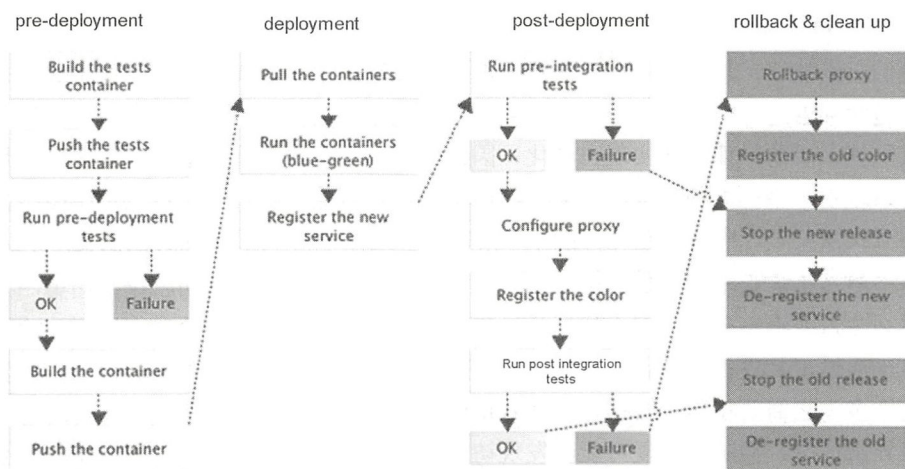


图 2-26 回滚与清理

8. 决定每个步骤的执行环境

决定每个步骤的执行环境是至关重要的。按照一般的规则来说，尽量不要在生产服务器中执行。这表示除了部署相关的任务都应当在一个专属于持续部署的独立的集群中执行。在图 2-27 中，我们将这些任务标记为黄色，并将需要在生产环境中执行的任务标记为蓝色。请注意，即使是蓝色的任务也不应当直接在生产环境中执行，而是通过工具的 API 执行。举例来说，如果使用 Docker Swarm 进行容器的部署，那么无须直接访问主节点所在的服务，而是创建 DOCKER_HOST 变量，将最终的目标地址通知本地 Docker 客户端。

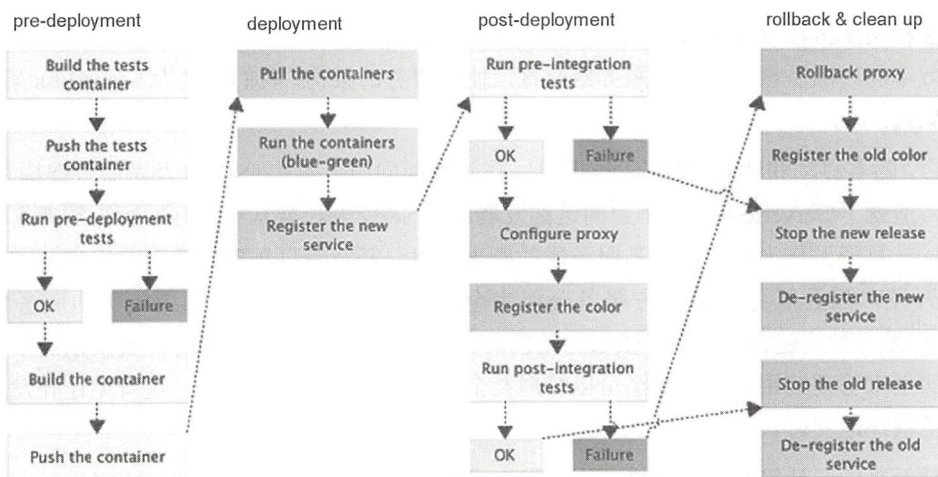


图 2-27 决定每个步骤的执行环境

9. 完成整个持续部署流

现在我们已经能够可靠地将每次签入部署至生产环境中了，但我们的工作只完成了一半。另一半工作是对部署进行监控，并根据实时数据与历史数据进行相应的操作。由于我们的最终目标是将代码签入后的一切操作实现自动化，因此人为的交互将会降至最低。创建一个具备自恢复能力的系统是一个很大的挑战，它需要我们持续进行调整。我们不仅希望系统能够从故障中恢复（响应式恢复），同时希望尽可能第一时间防止这些故障出现（预防性恢复）。

如果某个服务进程由于某种原因中止了运行，则系统应当再次将其初始化。如果产生故障的原因是某个节点变得不可靠，那么初始化过程应当在另一个健康的服务器中运行。响应式恢复的要点在于通过工具进行数据收集、持续地监控服务，并在发生故障时采取行动。预防性恢复则要复杂许多，它需要将历史数据记录在数据库中，对各种模式进行评估，以预测未来是否会发生某些异常情况。预防性恢复可能会发现访问量处于不断上升的情况，需要在几个小时之内对系统进行扩展。也可能每周一早上是访问量的峰值，系统在这段时间需要扩展，随后在访问量恢复正常之后收缩成原来的规模。

2.6 Serverless 架构

在目前主流云计算 IaaS（Infrastructure-as-a-Service，基础设施即服务）和 PaaS（Platform-as-a-Service，平台即服务）中，开发者进行业务开发时，仍然需要关心很多和服务器相关的服务端开发工作，比如缓存、消息服务、Web 应用服务器、数据库，以及对服务器进行性能优化，考虑存储和计算资源，考虑负载和扩展，考虑服务器容灾稳定性等非业务逻辑的开发。这些服务器的运维和开发，知识和经验极大地限制了开发者进行业务开发的效率。设想一下，如果开发者直接租用服务或开发服务而无须关注如何在服务器中运行部署服务，则是否可以极大地提升开发效率和产品质量？这种去服务器而直接使用服务的架构，我们称之为 Serverless 架构（无服务器架构）。

如今，随着移动和物联网应用蓬勃发展，伴随着面向服务架构（SOA）及微服务架构（MSA）的盛行，造就了 Serverless 架构平台的迅猛发展。在 Serverless 架构中，开发者无须考虑服务器的问题，计算资源作为服务而不是服务器的概念出现，开发者只需要关注面向客户的客户端业务程序开发，后台服务由第三方服务公司完全或部分提供，开发者调用相关的服务即可。Serverless 是一种构建和管理基于微服务架构的完整流程，允许我们在服务部署级别而不是服务器部署级别来管理应用部署，甚至可以管理某个具体功能或端口的部署，这就能让开发者快速迭代，更快速地交付软件。

这种新兴的云计算服务交付模式为开发人员和管理员带来了许多益处。它提供了合适的灵活性和控制性级别，因而在 IaaS 和 PaaS 之间找到了一条中间道路。由于服务器端几乎没有什

么要管理的，Serverless 架构正在彻底改变软件开发和部署领域，比如，推动了 NoOps 模式的发展。

2.6.1 什么是 Serverless 架构

Serverless 架构是新兴的架构体系，业界也没有一个明确的关于 Serverless 架构的定义。Mike Roberts 认为的 Serverless 架构主要有下面两种形式：

- 首先，Serverless 架构用于描述依赖第三方服务（云端）实现对逻辑和状态进行管理的应用。这些应用包括典型的富客户端应用，比如单页 Web 应用或移动应用，它们使用基于云的数据库（比如 Parse 或 Firebase），还有授权服务（比如 Auth0、AWS Cognito 等），这类服务以前曾经被描述为 BaaS（Backend as a Service，后端即服务）。
- 其次，Serverless 架构也可以指这样的一类应用，一部分服务逻辑由应用实现，但是跟传统架构不同在于，它们运行在无状态的容器中，可以由事件触发，短暂、完全地被第三方管理。一种观点认为这是 FaaS（Functions as service，函数服务），而 AWS Lambda 就是一种流行的 FaaS 实现，当然还有其他。

云计算的发展从 IaaS、PaaS、SaaS，到最新的 BaaS，在这个趋势中 Serverless（去服务器化）的趋势越来越明显。IaaS 将真实的物理机变成了虚拟机，PaaS 进一步将虚拟机变成包含基础设施的中间件服务。BaaS 和 SaaS 将中间件服务扩展到更基础的后端能力。这些是云计算解决效率和成本的重要体现。Serverless 这种无服务器架构，用服务代替服务器，无须了解落实服务，进一步提高了云计算的成本和效率，从而为 BaaS 这种新时代云计算提供了架构基础。

BaaS 要被开发者广泛接受，需要在云端解决以下的限制：

- BaaS 服务的治理；
- BaaS 服务需要提供逻辑定制扩展；
- BaaS 服务能独立部署，快速启动；
- BaaS 服务可以弹性扩展，满足高并发需要；
- BaaS 服务可以被监控、计费；
- BaaS 服务要解决 DevOps 相关的问题。

实现 Serverless 架构需要利用以下技术和方案：

- 实现 BaaS 中的云代码特性，开发者可以直接开发在云端的业务代码，实现 FaaS；
- 实现 API 网关，用 API 代表服务的入口，并对所有服务进行治理；
- 微服务架构技术，用微服务的概念来实施服务的开发；

- 利用 Docker 等容器技术部署运行微服务。

2.6.2 Serverless 典型的应用场景

1. UI 驱动的应用

先讨论一个带有服务功能逻辑的传统面向客户端的三层应用——一个典型的电子商务应用 Pet Store（在线宠物商店）。传统面向客户端的三层应用如图 2-28 所示，假设服务端用 Java 开发完成，客户端用 HTML/JavaScript 开发完成。



图 2-28 传统面向客户端的三层应用

在这种架构中，服务端不得不实现诸多系统逻辑，例如，认证、页面导航、搜索、交易等都需要在服务端完成，而客户端则显得相对比较简单。采用 Serverless 架构来对该应用进行改造，如图 2-29 所示。

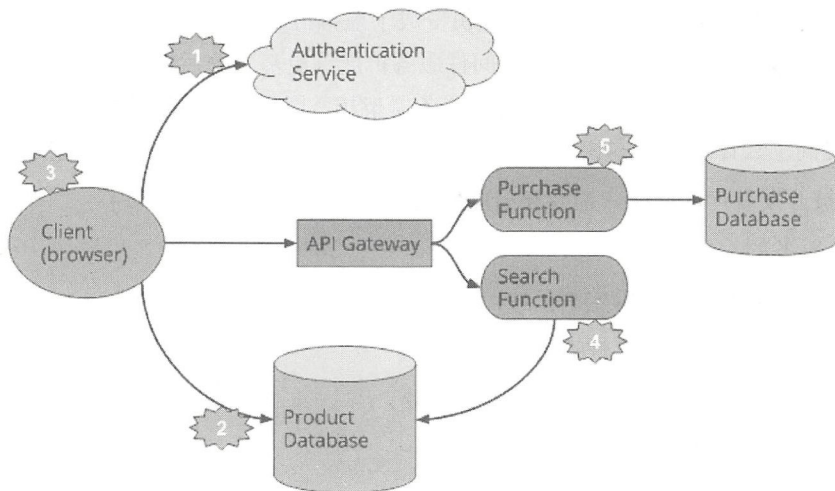


图 2-29 Serverless 架构的应用

Serverless 架构相比于传统面向客户端的三层应用架构，有以下几方面的差异：

- （1）删除了认证逻辑，用第三方 BaaS 服务来替代。
- （2）使用另外一个 BaaS，允许客户端直接访问架构与第三方（例如，AWS Dynamo）上

的数据子库。通过这种方式给客户提供更安全的访问数据库模式。

(3) 前两点包含很重要的第三点，也就是以前运行在 Pet Store 服务端的逻辑现在都转移到客户端中，例如，跟踪用户访问，理解应用的 UX 架构（如页面导航），读取数据库并转化为可视化视图等。客户端则慢慢转化为单页面应用。

(4) 某些我们想保留在服务端的 UX 相关功能，例如，计算敏感或需要访问大量数据，比如搜索这类应用。对于搜索这类需求，我们不需要运行一个专用服务，而是通过 API Gateway 对 HTTP 访问提供响应。这样可以使客户端和服务端都从同一个数据库中读取相关数据。由于原始服务使用 Java 开发，AWS Lambda (FaaS 提供者) 支持 Java 功能，因此可以直接从 Pet Store 服务端将代码移植到 Pet Store 搜索功能，而不用重写代码。

(5) 可以将“Purchase”功能用另外一个 FaaS 功能取代，因为安全原因放在服务端还不如在客户端重新实现，当然前端还是 API Gateway。

2. 消息驱动的应用

消息驱动的应用是一个纯后台数据处理服务，如图 2-30 所示。例如，正在编写一个面向用户的应用，需要对 UI 请求快速响应，同时还想获取所有发生的行为。我们设想一个 Ad Server（在线广告系统），当用户点击一个广告时，希望快速导向目标，同时需要搜集点击量以便向广告商收取费用。

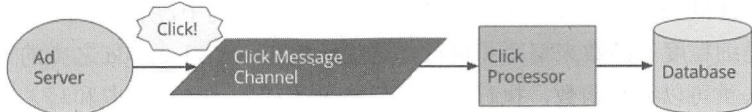


图 2-30 消息驱动的应用

在传统的架构中，Ad Server 同步地响应客户，同时还会向异步处理“点击量”的应用发送一个消息更新到便于以后向广告商收费的数据库。

而采用 Serverless 架构的情况下，将会改为如图 2-31 所示的方式。

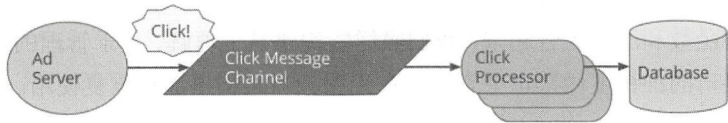


图 2-31 Serverless 的消息驱动的应用

这个架构跟第一个例子有些许不同，这里我们用 FaaS 功能取代了一个一直运行的应用。此

FaaS 运行于第三方提供商提供的消息驱动上下文之间。需要注意的是，供应商提供了消息代理和 FaaS，两者将更加紧密地合作在一起。

FaaS 环境通过复制出若干实例来并行处理这些点击，这无疑带来了全新开发体验和执行效率。

2.6.3 Serverless 架构原则

Peter Sbarski 和 Sam Kroonenburg 合著的 *Serverless Architectures on AWS* 一书中总结了 Serverless 架构的以下五种原则，运用这些原则可以帮助我们在构建 Serverless 架构应用时做出正确的决定：

- 根据需要，使用计算服务来执行代码（没有服务器）。
- 编写单一用途的无状态函数。
- 设计基于推送的、事件驱动的管道。
- 创建更强大的前端。
- 拥抱第三方服务。

接下来详细地分析每一个原则。

1. 根据需要，使用计算服务执行代码

Serverless 架构是 SOA 概念的自然延伸。在 Serverless 架构中，所有自定义代码作为孤立的、独立的、一般是细粒度的函数来编写和执行，这些函数在 AWS Lambda 之类的无状态计算服务中运行。开发人员可以编写函数，执行几乎任何常见的任务，比如读取和写入数据源，调用函数及执行计算。在比较复杂的情况下，开发人员可以构建更复杂的管道，编排多个函数的调用。可能会有这种场景：仍需要服务器来处理某个任务。然而，这种情况并不多见；作为开发人员，我们应该尽量避免运行服务器并与之交互。

2. 编写单一用途的无状态函数

作为软件工程师，我们在设计函数时应该尽量着眼于单一职责原则（SRP）。单单处理某一项任务的函数更容易测试、运行稳定，而且带来的错误和意外的副作用比较少。通过以一种松散编排的方式将函数和服务组合起来，我们照样能构建易于理解、易于管理的复杂后端系统。拥有明确定义的接口的细粒度函数也更有可能在无服务器架构里面被重复使用。

为 Lambda 等计算服务编写的代码应该以无状态方式来构建。它不得假设本地资源或进程会在当前的会话之后生存下去。无状态性功能很强大，因为它让平台得以迅速扩展，以处理数量不断变化的入站事件或请求。

3. 设计基于推送的、事件驱动的管道

可以构建满足任何用途的无服务器架构。系统可以一开始就构建成无服务器，也可以逐步重新设计现有的整体单一式应用程序，以便充分发挥这种架构的优势。最灵活、最强大的无服务器设计是事件驱动型的。

构建事件驱动的、基于推送的系统常常有望降低成本和复杂性（我们不需要运行额外代码来轮询变更），可能让用户体验更流畅。不言而喻，虽然事件驱动的、基于推送的模式是个美好的目标，但它们并非在所有情况下都是适当的或可以实现的。有时候，我们不得不实施一个 Lambda 函数来轮询事件源或按时间表运行。

4. 创建更强大的前端

有必要记住这一点，在 Lambda 中运行的自定义代码应该快速执行。早终结的函数较“便宜”，因为 Lambda 的定价基于请求数量、执行时间段及分配的内存量。在 Lambda 中要处理的事务比较少、比较“省钱”。此外，拥有调用服务的更丰富的前端有利于提供更好的用户体验。在线资源之间更少的环节和缩短的延迟会让人觉得应用程序的性能和可用性更好。

数字签名的令牌让前端可以与不同的服务（包括数据库）直接进行通信。相比之下，在传统系统中，所有通信经由后端服务器实现。让前端与服务进行通信有助于减少创建环节，能尽快获得所需的资源系统。

然而，不是一切服务都可以或都应该在前端执行。有些私密信息无法交给客户端设备来处理，比如，信用卡或向用户发送电子邮件只能由不受最终用户控制的服务来完成。在这种情况下，就需要计算服务来协调动作、验证数据，并实施安全。

另外要考虑的重要一点是一致性。如果前端负责写入多个服务，可是中途出现故障，则会导致系统处于不一致的状态。在这种情况下，应该使用 Lambda 函数，因为它可以从容地处理错误，重新尝试失败的操作。原子性和一致性在 Lambda 函数中实现起来和控制起来比在前端中容易得多。

5. 拥抱第三方服务

如果第三方服务能提供价值、减少自定义代码，则自然欢迎它们加入。如今，开发人员可以充分利用许多服务，从用于验证的 Auth0 服务，到用于支付处理的 Stripe 或 Braintree 服务，不一而足。考虑到价格、性能和可用性等因素，开发人员就应该试着采用第三方服务。开发人员花时间解决其领域独有的问题要比重复构建别人已经实施的功能有意义得多。如果已有了切实可行的第三方服务和 API，别纯粹为了构建而构建，应站在巨人的肩上达到新的高度。

2.6.4 例子：使用 Serverless 实现游戏全球同服

本例将演示利用 AWS 平台的 Serverless 架构来让游戏实现全球同服。

全球同服的游戏架构有如下的需求：

- 全球所有玩家的持久化信息（包括用户基本信息、等级、装备、进度等状态信息）都保存在中心站点。玩家统一通过 HTTP(S) 登录到中心站点并获取状态信息。
- 对战初始，由中心站点对玩家进行重定向到对应的 Game Server。在对战过程中使用 TCP 长连接从而保证更好的游戏体验。
- 对战结束后，客户端与 Game Server 中断 TCP 连接，对战结果数据回写到中心站点并保存最终的状态信息。

基于上述的架构，游戏完全构建在统一的“大世界”中（唯一中心站点），并且由分布在全球的 Game Server 来保证游戏的低延迟。由于 Game Server 分布在全球不同的地区，如何做到资源的快速扩展和按需伸缩将是一个难点。下面将以 Serverless 架构的方式实现这一需求。

首先，AWS 平台提供了非常完整的 API 接口，开发者可以选择各种语言的 SDK 完成对资源的调度，这里我们可以将代码运行在 Lambda 中。如下所示，我们的中心站点即 Lambda 部署的站点选择的是 Virginia（弗吉尼亚，美国东部地区），通过 Node.js SDK 跨地区到 Tokyo（东京，日本首都）来启动 EC2 服务器：

```
var AWS = require('aws-sdk');
exports.handler = function (event, context) {
  console.log("Received data as:", event);
  var ec2 = new AWS.EC2({region: 'ap-northeast-1'});
  var params = {
    ImageId: 'ami-29160d47',
    InstanceType: 't2.micro',
    KeyName: 'Tech-labs',
    SecurityGroupIds: ['sg-d0aalbb4'],
    IamInstanceProfile: {Name: 'EC2-Admin'},
    MinCount: 1,
    MaxCount: 1
  };

  // 创建实例
  ec2.runInstances(params, function (err, data) {
    if (err) {
```

```

        console.log("Could not create instance", err);
        context.fail(err);
    }
    var instanceId = data.Instances[0].InstanceId;
    console.log("Created instance", instanceId);

    // 存储实例 id 并设置状态
    context.succeed(instanceId);
});
};

```

由于启动 EC2 是一个异步过程，所以我们需要记录相关的服务器启动信息（InstanceId），并定义另一接口接收 Game Server 在服务就绪后返回的回执信息，代码如下：

```

var AWS = require('aws-sdk');
exports.handler = function (event, context) {
    console.log("Received data as:", event);
    var instanceId = event.instanceId;
    var region = event.region;
    var publicIp = event.publicIp;
    var version = event.version;
    ...

    // 检查 instanceId 并在线更新实例状态
};

```

同时，这种回执接口的 API（包括其他 API）都可以考虑使用 Amazon API Gateway 服务进行部署。API Gateway 可以帮助我们将现有函数快速发布为 RESTful 的 API 接口，并同时利用 CloudFront 的边缘节点进行部署，以保证访问端能获得更低的延迟。按照上例的回执，Lambda 函数可以构造 API Gateway 的配置，如图 2-32 所示。

请求示例：

```

/game/servers/i-dd861842

{
  "region": "ap-northeast-1",
  "publicIp": "52.193.34.102",
  "version": "110"
}

```

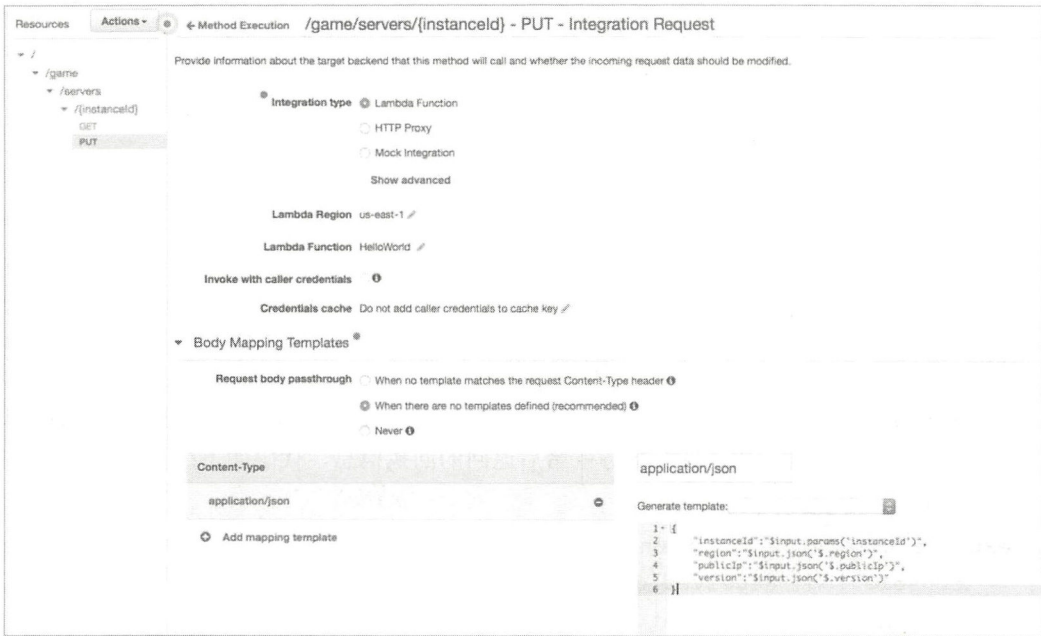


图 2-32 API Gateway 的配置

接下来，为了确保 Game Server 的状态是正常的，使玩家能被路由到正确的服务器上，可以构造另一个类似心跳的 Lambda 函数，用来接收 Game Server 的状态信息，心跳频率可根据需求进行调整。当然，在频率不需要很高的情况下（ ≥ 1 分钟），也可以利用 CloudWatch 来发起报警，并同时发起 SNS 通知 Lambda 函数以更新 Game Server 的状态。

在 Game Server 具备自动按需扩展（Scale out）的能力后，我们就需要考虑如何解决 Game Server 的缩减（Scale in）了。这里我们采用 CloudWatch→SNS→Lambda (cross region) 的方式来实现 Game Server 的缩减，具体流程如下：

（1）Game Server 自定义指标（custom metrics）将当前服务器的在线人数发送到 CloudWatch 中。

```
#!/bin/bash
#get instance-id from local meta-data id=$(curl -s http://169.254.169.254/latest/meta-data/instance-id)
#get current onlinePlayers players=$(...) aws cloudwatch put-metric-data --metric-name "OnlinePlayers" --namespace "GameServer" --dimension InstanceId=$id --value $players
```

（2）设定 CloudWatch 的报警规则，当服务器在线人数为零时，会触发 SNS 通知，如图 2-33 所示。

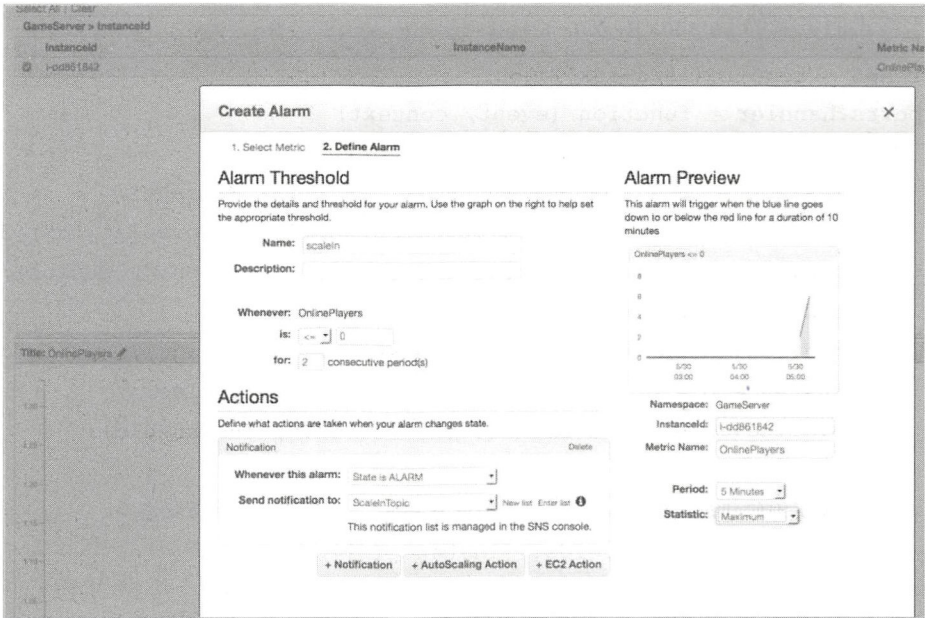


图 2-33 CloudWatch 自定义指标报警

在实际场景中，需要通过如下脚本自动建立报警：

```
aws cloudwatch put-metric-alarm --alarm-name $id-players--alarm-description
"Alarm when online players less than 0" --metric-name "OnlinePlayers" --namespace
"GameServer" --dimension "Name=InstanceId,Value=$id" --statistic Maximum
--period 300 --threshold 0 --comparison-operator LessThanOrEqualToThreshold
--evaluation-periods 2 --alarm-actions arn:aws:sns:ap-northeast-1:11111111222:
ScaleInTopic
```

(3) 订阅了 SNS 通知主题的中心站点的 Lambda 函数用于把机器终止，如图 2-34 所示。

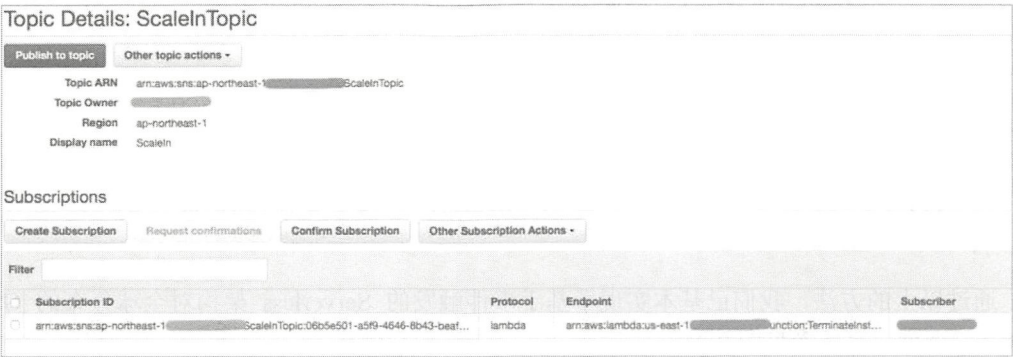


图 2-34 Lambda 函数订阅 SNS 服务通知

用于终止服务器的 Lambda 函数：

```
var AWS = require('aws-sdk');
exports.handler = function (event, context) {
    console.log("Received data as:", event);
    var message = JSON.parse(event.Records[0].Sns.Message);
    var region = event.Records[0].EventSubscriptionArn.split(":")[3];
    console.log("Need to terminate the server in region:", region);
    var ec2 = new AWS.EC2({region: region});
    console.log("Need to terminate the server:", message);
    var instanceId = message.Trigger.Dimensions[0].value;
    console.log("Need to terminate the server:", instanceId);

    // 检查实例的状态是否可以从 DynamoDB 中终止，并在终止时更新状态
    var params = {InstanceIds: [instanceId]};

    // 终止实例
    ec2.terminateInstances(params, function (err, data) {
        if (err) {
            console.log("Could not terminate instance", err);

            // 回滚终止的实例
            context.fail(err);
        }
        for (var i in data.TerminatingInstances) {
            var instance = data.TerminatingInstances[i];
            console.log('TERM:\t' + instance.InstanceId);

            // 删除终止的实例
        }
        context.succeed(data.TerminatingInstances);
    });
};
```

通过以上的方 法，我们已基本实现了基于事件触发的 Serverless 架构对全球分布的 Game Server 的调度，具体参见图 2-35。

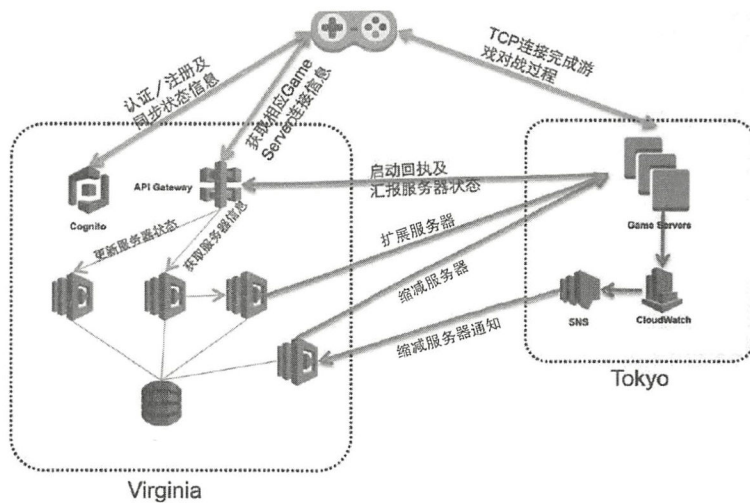
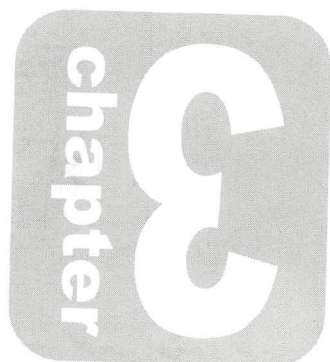


图 2-35 Serverless 全球同服游戏架构

第 3 章

分布式消息服务



3.1 分布式消息概述

在 SOA 或微服务架构中，普遍会采用 HTTP 协议作为通信协议。HTTP 协议具有平台无关性、语言中立性等特点，在分布式系统中广泛应用。特别是微服务架构的流行，遵循一致的 REST 风格的 HTTP 协议更能在各个微服务之间实现低沟通成本的通信。然而，HTTP 有一个缺点，就是它的请求是同步的，即遵循的是“请求—响应”模式，在服务器未返回结果之前，HTTP 客户端会一直等待，直到获取结果或超时，这在一定程度上限制了程序的处理能力，毕竟等待就是浪费。

消息中间件正好弥补了上述 HTTP 协议的不足。消息中间件往往会支持多种语言的客户端（比如 Java、C、C++、C#、Ruby 等），支持多种协议（HTTP、TCP、SSL、NIO、UDP 等）。消息中间件支持异步通信，从而可以极大地提升通信效率。

3.1.1 基本概念

消息中间件的基本原理十分简单，就是接收和转发消息。比如邮局场景：当你将一个包裹送到邮局时，邮递员会将邮件送到收件人手上。消息中间件就好比一个邮递员。

目前市面上流行的消息中间件往往具备以下几个基本的概念：

- Topic（主题）——按照分类对信息源进行维护。实际应用中一个业务一个 Topic。
- Producer（生产者）——把消息发送到 Topic 中的进程叫作生产者。
- Consumer（消费者）——把从 Topic 中订阅消息的进程叫作消费者。
- Broker（服务）——集群中的每个服务叫作 Broker。

上述概念在不同的产品中可能有不同的表述，但所承担的功能都是类似的。

3.1.2 使用场景

如果 HTTP 协议能满足业务需要，则首先应该选择使用 HTTP 协议作为服务间的通信协议。如果 HTTP 不能满足，则选用消息中间件产品作为补充，往往可以获得以下收益：

- 异步通信。异步意味着程序在处理结果完成之前无须等待就可以去干其他事情，避免了资源的浪费。
- 解耦。生产者把消息发送到消息队列中，这个过程就结束了。至于谁会从消息队列中去取消息、消费消息，生产者是不需要关心的。这样就实现了生产者和消费者的解耦。

- 数据缓冲。当有消息队列接收大量消息时，会先缓存到消息队列中，从而避免了由于消息处理能力不足而导致的程序崩溃。
- 多种消息推送模型。消息中间件一般都会支持 Publish/Subscribe 和 P2P 等消息模型，以满足各种使用场景的需要。
- 强顺序。消息在消息中间件中按照可靠的 FIFO 和严格的通信顺序来进行消费。这在某些需要强顺序要求的场景中非常有用，比如事务处理、事件通知等。
- 持久化消息。消息中间件能够安全地保存消息，直到消费者收到消息。
- 支持分布式。消息中间件往往支持分布式部署，具有高可用、高并发的能力。

3.1.3 常用技术

目前，市面上流行的消息中间件产品很多，成熟的开源产品也数不胜数。比如，老牌的产品 RabbitMQ 以高效而著称；Apache Kafka 能够支持各种强大的消息模式，从而被互联网公司广泛采用；Apache ActiveMQ 是用 Java 语言编写的，能够支持全面的 JMS 和 J2EE 规范；RocketMQ 则是来自阿里巴巴的“国货精品”，目前已经属于 Apache 基金会管理，算是走出了国门。不同的中间件产品有各自的优缺点。选择一款适合自己项目的中间件，需要花時間を进行评估。在接下来的章节，也会对上述提到的产品进行详细的介绍。

3.2 Apache ActiveMQ

Apache ActiveMQ 支持多种语言客户端（Java、C、C++、C#、Ruby 等），支持多种协议（HTTP、TCP、SSL、NIO、UDP 等）及良好的 Spring 支持。

3.2.1 例子：producer-consumer

本例将演示在 producer-consumer（生产者—消费者）模式下，通过 Broker 生产和消费消息。

1. 准备

为了演示该例子，先要运行 JMS broker。在命令行 shell 中执行如下指令：

```
bin/activemq console
```

这样就启动了 ActiveMQ。

2. 运行

通过命令行来运行：

```
${ACTIVEMQ_HOME}/bin/activemq producer  
${ACTIVEMQ_HOME}/bin/activemq consumer
```

如果存在 `activemq-all` 的 jar 包，则使用下面的命令能达到相同的效果：

```
java -jar activemq-all-5.x.x.jar producer  
java -jar activemq-all-5.x.x.jar consumer
```

如果在 `Kafka` 内部运行，则执行下面的语句：

```
activemq:producer  
activemq:consumer
```

更多运行时的参数设置，可以使用 `--help` 来查看。

3. 生产和消费消息

发送自定义的文本消息：

```
bin/activemq producer --message "My message" --messageCount 1
```

发送自定义长度的字节消息：

```
bin/activemq producer --messageSize 100 --messageCount 1
```

发送文本消息，从 URL 获取的内容如下：

```
bin/activemq producer --payloadUrl http://activemq.apache.org/schema/core/  
activemq-core.xsd --messageCount 1
```

事务（`transaction`）模式下的消费消息：

```
bin/activemq consumer --transacted true
```

使用 `client acknowledgment` 模式：

```
bin/activemq consumer --ackMode CLIENT_ACKNOWLEDGE
```

使用持久主题订阅者（`durable topic subscriber`）：

```
bin/activemq consumer --durable true --clientId example --destination topic://TEST
```

3.2.2 例子：使用 JMX 来监控 ActiveMQ

Apache ActiveMQ 对 JMX 进行了扩展，允许通过 JMX MBeans 来监控和控制 broker 的行为。

1. 修改配置，运行 Broker

修改 activemq.xml 配置文件，确保 useJmx 属性为 true（默认是 true），managementContext 的 createConnector 属性为 true（默认是 false）。

例如：

```
<broker useJmx="true" brokerName="BROKER1">
...
  <managementContext>
    <managementContext createConnector="true"/>
  </managementContext>
...
</broker>
```

2. 运行 JMX 控制台

```
$ jconsole
```

3. 连接到指定的 JMX URL

可以在本地进程列表中看到 ActiveMQ broker，访问该进程。

也可以指定 JMX URL 来进行远程访问（默认的连接账号和密码为 admin/activemq），格式如下：

```
service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi
```

JMX 访问 ActiveMQ 进程如图 3-1 所示。



图 3-1 JMX 访问 ActiveMQ 进程



在 Windows 环境下, JMX 控制台的界面如图 3-2 所示。

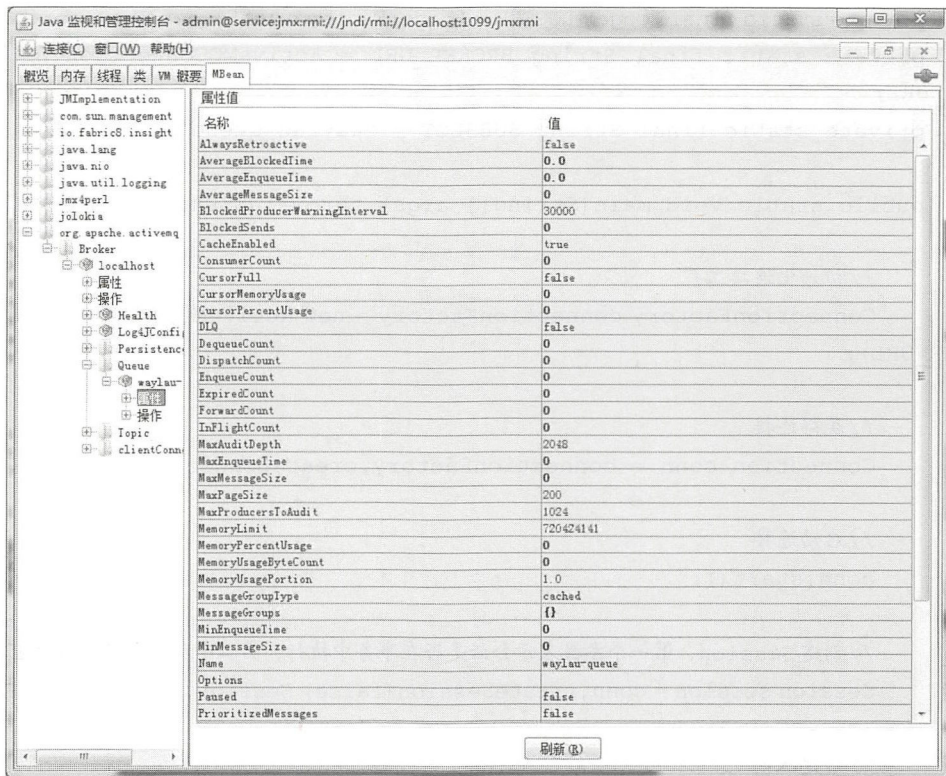


图 3-2 JMX 控制台的界面

4. 配置 JMX 控制台的账号和密码

在 `conf/jmx.access` 中配置用户账号及访问权限, 如:

```
monitorRole readonly
controlRole readwrite
```

在 `conf/jmx.password` 中配置用户账号的密码, 如:

```
monitorRole abc123
controlRole abcd1234
```

3.2.3 例子: 使用 Java 实现 producer-consumer

生产者程序 `Producer.java`:

```
public class Producer {
```



```
private static final Logger LOGGER = LoggerFactory.getLogger(Producer.class);
private static final String BROKER_URL = ActiveMQConnection.DEFAULT_
BROKER_URL;
private static final String SUBJECT = "waylau-queue";

public static void main(String[] args) throws JMSEException {

    //初始化连接工厂
    ConnectionFactory connectionFactory = new ActiveMQConnectionFactory
(BROKER_URL);

    //获得连接
    Connection conn = connectionFactory.createConnection();

    //启动连接
    conn.start();

    //创建 Session, 第一个参数表示会话是否在事务中执行, 第二个参数设定会话的应答模式
    Session session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);

    //创建队列
    Destination dest = session.createQueue(SUBJECT);

    //createTopic 方法用来创建 Topic
    //session.createTopic("TOPIC");

    //通过 session 可以创建消息的生产者
    MessageProducer producer = session.createProducer(dest);
    for (int i=0;i<100;i++) {

        //初始化一个 MQ 消息
        TextMessage message = session.createTextMessage("Welcome to
waylau.com " + i);

        //发送消息
        producer.send(message);
    }
}
```



```
        LOGGER.info("send message {}", i);
    }

    //关闭 MQ 连接
    conn.close();
}
}
```

消费者程序 `Producer.java`:

```
public class Consumer implements MessageListener {

    private static final Logger LOGGER = LoggerFactory.getLogger(Consumer.class);
    private static final String BROKER_URL = ActiveMQConnection.DEFAULT_
BROKER_URL;
    private static final String SUBJECT = "waylau-queue";

    public static void main(String[] args) throws JMSEException {

        //初始化 ConnectionFactory
        ConnectionFactory connectionFactory = new ActiveMQConnectionFactory
(BROKER_URL);

        //创建 MQ 连接
        Connection conn = connectionFactory.createConnection();
        //启动连接
        conn.start();

        //创建会话
        Session session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);

        //通过会话创建目标
        Destination dest = session.createQueue(SUBJECT);

        //创建 MQ 消息的消费者
        MessageConsumer consumer = session.createConsumer(dest);

        //初始化 MessageListener
```





```

Consumer me = new Consumer();

//给消费者设定监听对象
consumer.setMessageListener(me);
}

@Override
public void onMessage(Message message) {
    TextMessage txtMessage = (TextMessage)message;
    try {
        LOGGER.info ("get message " + txtMessage.getText());
    } catch (JMSEException e) {
        LOGGER.error("error {}", e);
    }
}
}

```

1. 执行命令来启动 ActiveMQ

bin/activemq start

2. 生产者执行

执行如下命令：

```
mvn clean compile exec:java -Dexec.mainClass=com.waylau.activemq.ProducerApp
```

输出如下：

```

20:12:10.807 [ActiveMQ Task-1] INFO  org.apache.activemq.transport.
failover.FailoverTransport-Successfully connected to tcp://localhost:61616
20:12:10.928 [main] INFO  com.waylau.activemq.Producer - send message 0
20:12:10.963 [main] INFO  com.waylau.activemq.Producer - send message 1
20:12:10.992 [main] INFO  com.waylau.activemq.Producer - send message 2
20:12:11.019 [main] INFO  com.waylau.activemq.Producer - send message 3
20:12:11.036 [main] INFO  com.waylau.activemq.Producer - send message 4
20:12:11.058 [main] INFO  com.waylau.activemq.Producer - send message 5
20:12:11.085 [main] INFO  com.waylau.activemq.Producer - send message 6
20:12:11.113 [main] INFO  com.waylau.activemq.Producer - send message 7
20:12:11.141 [main] INFO  com.waylau.activemq.Producer - send message 8

```





```
20:12:11.191 [main] INFO com.waylau.activemq.Producer - send message 9
```

3. 消费者执行

执行如下命令：

```
mvn clean compile exec:java -Dexec.mainClass=com.waylau.activemq.ConsumerApp
```

输出如下：

```
20:12:05.262 [ActiveMQ Task-1] INFO org.apache.activemq.transport.  
failover.FailoverTransport-Successfully connected to tcp://localhost:61616  
20:12:10.875 [ActiveMQ Session Task-1] INFO com.waylau.activemq.Consumer  
- get message Welcome to waylau.com 0  
20:12:10.939 [ActiveMQ Session Task-1] INFO com.waylau.activemq.Consumer  
- get message Welcome to waylau.com 1  
20:12:10.965 [ActiveMQ Session Task-1] INFO com.waylau.activemq.Consumer  
- get message Welcome to waylau.com 2  
20:12:10.994 [ActiveMQ Session Task-1] INFO com.waylau.activemq.Consumer  
- get message Welcome to waylau.com 3  
20:12:11.020 [ActiveMQ Session Task-1] INFO com.waylau.activemq.Consumer  
- get message Welcome to waylau.com 4  
20:12:11.038 [ActiveMQ Session Task-1] INFO com.waylau.activemq.Consumer  
- get message Welcome to waylau.com 5  
20:12:11.059 [ActiveMQ Session Task-1] INFO com.waylau.activemq.Consumer  
- get message Welcome to waylau.com 6  
20:12:11.086 [ActiveMQ Session Task-1] INFO com.waylau.activemq.Consumer  
- get message Welcome to waylau.com 7  
20:12:11.114 [ActiveMQ Session Task-1] INFO com.waylau.activemq.Consumer  
- get message Welcome to waylau.com 8  
20:12:11.142 [ActiveMQ Session Task-1] INFO com.waylau.activemq.Consumer  
- get message Welcome to waylau.com 9
```

4. 控制台可以监控 ActiveMQ 的状态

使用浏览器访问 <http://127.0.0.1:8161/admin/>，账号和密码都是 admin，可以看到监控界面，上面统计了每个队列的消费者数、生产者数，以及发送的消息数等数据。

上面例子中的代码，可以在 <https://github.com/waylau/distributed-systems-technologies-and-cases-analysis> 的 activemq-demo 程序中找到。





3.3 RabbitMQ

RabbitMQ 由 Erlang 语言写成，以高性能、健壮及可伸缩性著称。

3.3.1 例子：Work Queues

Work Queues（工作队列）又叫作 Task Queues（任务队列），背后主要的思想是避免立即处理一个资源密集型任务造成的长时间等待。相反我们可以计划着让任务后续执行。我们将任务封装成消息发送到队列中。一个 Worker（工作者）进程在后台运行，获取任务并最终执行任务。当运行多个 Worker（工作者）时，所有的任务将会被它们所共享。

图 3-3 是 Work Queues 的示意图。

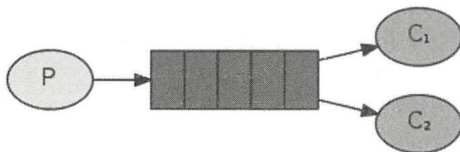


图 3-3 Work Queues 示意图

本节中的例子都采用 Java 语言编写，若熟悉 Python 语言，推荐阅读 Alvaro Videla 和 Jason J. W. Williams 合著的《RabbitMQ 实战》（该书中文版由汪佳南翻译，电子工业出版社出版）。

1. NewTask.java 程序

NewTask 是发送消息的任务发送程序。NewTask.java 代码如下：

```
public class NewTask {

    private static final Logger LOGGER = LoggerFactory.getLogger(NewTask.class);
    private static final String TASK_QUEUE_NAME = "waylau_queue";

    public static void main(String[] argv) throws Exception {

        // 初始化连接工厂
        ConnectionFactory factory = new ConnectionFactory();

        // 设置连接的地址
        factory.setHost("localhost");

        // 获得连接
```





```
Connection connection = factory.newConnection();

// 创建 Channel
Channel channel = connection.createChannel();

// 声明队列
channel.queueDeclare(TASK_QUEUE_NAME, true, false, false, null);

// 从控制台参数中, 获取消息
String message = getMessage(argv);

// 发送 10 条消息
for (int i = 0; i < 10; i++) {
    String msg = message + " " + i;
    channel.basicPublish("", TASK_QUEUE_NAME, MessageProperties.
        PERSISTENT_TEXT_PLAIN,
        (msg).getBytes("UTF-8"));

    LOGGER.info(" [x] Sent '" + msg + "'");
}

channel.close();
connection.close();
}

private static String getMessage(String[] strings) {
    if (strings.length < 1)
        return "Hello World!";
    return joinStrings(strings, " ");
}

private static String joinStrings(String[] strings, String delimiter) {
    int length = strings.length;
    if (length == 0)
        return "";
    StringBuilder words = new StringBuilder(strings[0]);
    for (int i = 1; i < length; i++) {
        words.append(delimiter).append(strings[i]);
    }
}
```





```
    }  
    return words.toString();  
}  
}
```

2. Worker.java 程序

Worker 是接收消息的工作者。Worker.java 代码如下：

```
public class Worker {  
  
    private static final Logger LOGGER = LoggerFactory.getLogger(Worker.class);  
    private static final String TASK_QUEUE_NAME = "waylau_queue";  
  
    public static void main(String[] argv) throws Exception {  
  
        // 初始化连接工厂  
        ConnectionFactory factory = new ConnectionFactory();  
  
        // 设置连接的地址  
        factory.setHost("localhost");  
  
        // 获得连接  
        final Connection connection = factory.newConnection();  
  
        // 创建 Channel  
        final Channel channel = connection.createChannel();  
  
        // 声明队列  
        channel.queueDeclare(TASK_QUEUE_NAME, true, false, false, null);  
        channel.basicQos(1);  
  
        LOGGER.info(" [*] Waiting for messages. To exit press CTRL+C");  
  
        // 消费者  
        final Consumer consumer = new DefaultConsumer(channel) {  
            @Override  
            public void handleDelivery(String consumerTag, Envelope envelope,  
                AMQP.BasicProperties properties,  
                byte[] body) throws IOException {
```





```
String message = new String(body, "UTF-8");

LOGGER.info(" [x] Received '" + message + "'");
try {
    doWork(message);
} finally {
    LOGGER.info(" [x] Done");

    // 确认消息
    channel.basicAck(envelope.getDeliveryTag(), false);
}
}

// 取消 autoAck
channel.basicConsume(TASK_QUEUE_NAME, false, consumer);
}

private static void doWork(String task) {
    for (char ch : task.toCharArray()) {
        if (ch == '.') {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException _ignored) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
}
```

3. 启动程序

先启动 RabbitMQ 服务器，再使用 Maven 语句 `mvn clean package` 编译程序。编译后，会在 `target` 目录下生成可执行的 `rabbitmq-demo-1.0.0.jar` 文件。

启动 Worker 执行如下：

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.Worker
```

我们为了演示效果，这里启动了 2 个 Worker。



启动 NewTask 执行如下：

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.NewTask welcom  
to waylau.com
```

NewTask 执行输出如下：

```
14:48:46.957 [main] INFO com.waylau.rabbitmq.NewTask - [x] Sent 'welcome  
to waylau.com 0'  
14:48:46.961 [main] INFO com.waylau.rabbitmq.NewTask - [x] Sent 'welcome  
to waylau.com 1'  
14:48:46.962 [main] INFO com.waylau.rabbitmq.NewTask - [x] Sent 'welcome  
to waylau.com 2'  
14:48:46.964 [main] INFO com.waylau.rabbitmq.NewTask - [x] Sent 'welcome  
to waylau.com 3'  
14:48:46.966 [main] INFO com.waylau.rabbitmq.NewTask - [x] Sent 'welcome  
to waylau.com 4'  
14:48:46.969 [main] INFO com.waylau.rabbitmq.NewTask - [x] Sent 'welcome  
to waylau.com 5'  
14:48:46.970 [main] INFO com.waylau.rabbitmq.NewTask - [x] Sent 'welcome  
to waylau.com 6'  
14:48:46.973 [main] INFO com.waylau.rabbitmq.NewTask - [x] Sent 'welcome  
to waylau.com 7'  
14:48:46.975 [main] INFO com.waylau.rabbitmq.NewTask - [x] Sent 'welcome  
to waylau.com 8'  
14:48:46.976 [main] INFO com.waylau.rabbitmq.NewTask - [x] Sent 'welcome  
to waylau.com 9'
```

第一个 Worker 执行输出如下：

```
14:48:40.256 [main] INFO com.waylau.rabbitmq.Worker - [*] Waiting for  
messages. To exit press CTRL+C  
14:48:46.952 [pool-2-thread-4] INFO com.waylau.rabbitmq.Worker - [x]  
Received'welcome to waylau.com 0'  
14:48:47.953 [pool-2-thread-4] INFO com.waylau.rabbitmq.Worker - [x] Done  
14:48:47.954 [pool-2-thread-5] INFO com.waylau.rabbitmq.Worker - [x]  
Received'welcome to waylau.com 2'  
14:48:48.955 [pool-2-thread-5] INFO com.waylau.rabbitmq.Worker - [x] Done  
14:48:48.957 [pool-2-thread-6] INFO com.waylau.rabbitmq.Worker - [x]  
Received'welcome to waylau.com 4'  
14:48:49.958 [pool-2-thread-6] INFO com.waylau.rabbitmq.Worker - [x] Done
```



```

14:48:49.959 [pool-2-thread-7] INFO com.waylau.rabbitmq.Worker - [x]
Received'welcome to waylau.com 6'
14:48:50.960 [pool-2-thread-7] INFO com.waylau.rabbitmq.Worker - [x] Done
14:48:50.961 [pool-2-thread-8] INFO com.waylau.rabbitmq.Worker - [x]
Received'welcome to waylau.com 8'
14:48:51.965 [pool-2-thread-8] INFO com.waylau.rabbitmq.Worker - [x] Done

```

第二个 Worker 执行输出如下:

```

14:48:44.371 [main] INFO com.waylau.rabbitmq.Worker - [*] Waiting for
message. To exit press CTRL+C
14:48:46.967 [pool-2-thread-4] INFO com.waylau.rabbitmq.Worker - [x]
Received'welcome to waylau.com 1'
14:48:47.968 [pool-2-thread-4] INFO com.waylau.rabbitmq.Worker - [x] Done
14:48:47.969 [pool-2-thread-5] INFO com.waylau.rabbitmq.Worker - [x]
Received'welcome to waylau.com 3'
14:48:48.970 [pool-2-thread-5] INFO com.waylau.rabbitmq.Worker - [x] Done
14:48:48.972 [pool-2-thread-6] INFO com.waylau.rabbitmq.Worker - [x]
Received'welcome to waylau.com 5'
14:48:49.973 [pool-2-thread-6] INFO com.waylau.rabbitmq.Worker - [x] Done
14:48:49.976 [pool-2-thread-7] INFO com.waylau.rabbitmq.Worker - [x]
Received'welcome to waylau.com 7'
14:48:50.977 [pool-2-thread-7] INFO com.waylau.rabbitmq.Worker - [x] Done
14:48:50.979 [pool-2-thread-8] INFO com.waylau.rabbitmq.Worker - [x]
Received'welcome to waylau.com 9'
14:48:51.980 [pool-2-thread-8] INFO com.waylau.rabbitmq.Worker - [x] Done

```

可以看到, NewTask 发送了 10 条消息, 而两个 Worker 在接收消息时, 对消息进行了均衡处理, 这在消息队列里面称为 Round-robin dispatching (循环分派)。这就是使用任务队列的优势之一, 让任务可以很容易地并行处理。如果我们正在处理一些堆积的文件, 则仅需要增加更多的工作者即可, 通过这种方式实现程序的性能扩展。

4. 程序详解

在声明队列时, `channel.queueDeclare(TASK_QUEUE_NAME, true, false, false, null)` 中 `queueDeclare` 方法的第二个参数是用来说明该队列是持久化的。这样可以保证即使 RabbitMQ 重启, 队列也不会丢失。现在我们需要标记消息持久化, 设置 `MessageProperties` (实现了 `BasicProperties`) 的值为 `PERSISTENT_TEXT_PLAIN` 即可。

`channel.basicQos(1)` 将告知 RabbitMQ 不要同时给一个工作者超过一个任务, 换句话说, 在



一个工作者完成处理任务，发送确认之前不要给它分发一个新的消息。取而代之的是把消息分发给下一个不繁忙的工作者。这样就实现了消息处理的负载均衡。

RabbitMQ 通过“Message acknowledgment（消息确认）”机制来保证消息已经被送达。一个消息确认是由消费者发出的，告诉 RabbitMQ 这个消息已经被接收，处理完成后，RabbitMQ 就可以删除它了。如果一个消费者没有发送确认信号，则 RabbitMQ 将认定这个消息没有完全处理成功，会把它传递给另一个消费者。通过这种方式，即使 Worker（工作者）有时会“死掉”，依旧可以保证没有消息会丢失。这里不存在消息超时的情况，RabbitMQ 只会在 worker 连接“死掉”后才重新传递这个消息。即使一个消息要被处理很长时间，也不是问题。消息确认机制默认情况下是开着的。首先，将 `channel.basicConsume(TASK_QUEUE_NAME, false, consumer)` 的第二个参数设置为 `false`，意味着不需要自动询问。其次，显式执行 `channel.basicAck(envelope.getDeliveryTag(), false)` 来提交询问。

3.3.2 例子：Publish/Subscribe

Publish/Subscribe（发布/订阅）在消息队列中是一种比较常见的工作模型，如图 3-4 所示。该模式定义了如何向一个内容节点发布和订阅消息，内容节点也叫主题（Topic），主题是为发布者（Publisher）和订阅者（Subscribe）提供传输的中介。Publish/Subscribe 模型使发布者和订阅者之间不需要直接通信（如 RMI）就可保证消息的传送，有效解决了系统间的耦合问题。该模式也定义了一种一对多的依赖关系，让多个订阅者对象同时监听某一个主题对象。

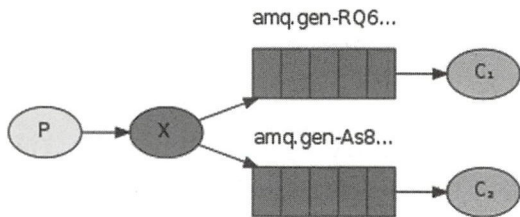


图 3-4 Publish/Subscribe 示意图

在该模式中，producer（生产者）并不直接发送消息到 queue（队列），而是发到 exchange（交换器）中。exchange 一边接收来自 producer 的消息，另外一边将消息插入 queue 中。在上一例子中，我们并没有显式地使用 exchange，仍然可以发送和接收消息。这是因为我们使用了一个默认的转发器，它的标识符为“”。之前发送消息的代码如下：

```
channel.basicPublish("",
    QUEUE_NAME, MessageProperties.PERSISTENT_TEXT_PLAIN, message.
getBytes());
```



这种没有显式命名的 exchange 被称为 nameless exchange（无名交换器）。

exchange 必须清楚接收到消息的下一步用途，比如，是要将消息插入一个 queue 中还是插入多个 queue 中。exchange type（交换器类型）标识的消息的用途，主要有 direct、topic、headers 和 fanout。比如，我们要创建一个 fanout 类型的 exchange，可以使用如下方法：

```
channel.exchangeDeclare("logs", "fanout");
```

fanout 类型的 exchange 特别简单——把所有它接收的消息广播到所有它知道的队列。

1. EmitLog.java 程序

EmitLog.java 程序用于日志发送。EmitLog.java 代码如下：

```
public class EmitLog {

    private static final Logger LOGGER = LoggerFactory.getLogger(EmitLog.class);

    private static final String EXCHANGE_NAME = "logs";

    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        // 声明交换器和类型
        channel.exchangeDeclare(EXCHANGE_NAME, "fanout");

        String message = getMessage(argv);

        // 往交换器上发送消息
        channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes("UTF-8"));
        LOGGER.info(" [x] Sent '" + message + "'");

        channel.close();
        connection.close();
    }

    private static String getMessage(String[] strings) {
        if (strings.length < 1)
            return "No message text";
        return strings[0];
    }
}
```

```
        return "info: Hello World!";
    }
    return joinStrings(strings, " ");
}

private static String joinStrings(String[] strings, String delimiter) {
    int length = strings.length;
    if (length == 0)
        return "";
    StringBuilder words = new StringBuilder(strings[0]);
    for (int i = 1; i < length; i++) {
        words.append(delimiter).append(strings[i]);
    }
    return words.toString();
}
}
```

2. ReceiveLogs.java 程序

ReceiveLogs.java 程序用于日志接收。ReceiveLogs.java 代码如下：

```
public class ReceiveLogs {

    private static final Logger LOGGER = LoggerFactory.getLogger(ReceiveLogs.
class);

    private static final String EXCHANGE_NAME = "logs";

    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        channel.exchangeDeclare(EXCHANGE_NAME, "fanout");

        // 不传递任何参数来创建一个非持久的、唯一的、自动删除的队列，该队列名称由服务器随机产生
        String queueName = channel.queueDeclare().getQueue();

        // 为交换器指定队列，设置 binding
        channel.queueBind(queueName, EXCHANGE_NAME, "");
    }
}
```

```

    LOGGER.info(" [*] Waiting for messages. To exit press CTRL+C");

    Consumer consumer = new DefaultConsumer(channel) {
        @Override
        public void handleDelivery(String consumerTag, Envelope envelope,
            AMQP.BasicProperties properties,
            byte[] body) throws IOException {
            String message = new String(body, "UTF-8");
            LOGGER.info(" [x] Received '" + message + "'");
        }
    };
    channel.basicConsume(queueName, true, consumer);
}
}

```

3. 启动程序

先启动 RabbitMQ 服务器。再使用 Maven 语句: `mvn clean package` 编译程序。编译后, 会在 `target` 目录下生成可执行的 `rabbitmq-demo-1.0.0.jar` 文件。

启动 `ReceiveLogs` 执行如下:

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.ReceiveLogs
```

为了演示效果更好, 启动了 2 个 `ReceiveLogs`。

启动 `EmitLog` 如下:

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.EmitLog welcom
to waylau.com
```

`EmitLog` 执行输出如下:

```
23:39:34.658 [main] INFO com.waylau.rabbitmq.EmitLog - [x] Sent 'welcom
to waylau.com'
```

两个 `ReceiveLogs` 都收到了相同的消息, 如下:

```
23:39:22.038 [main] INFO com.waylau.rabbitmq.ReceiveLogs - [*] Waiting
for messages. To exit press CTRL+C
23:39:34.660 [pool-2-thread-4] INFO com.waylau.rabbitmq.ReceiveLogs - [x]
Received 'welcom to waylau.com'
```

4. 程序详解

在 `ReceiveLogs` 程序中，`String queueName = channel.queueDeclare().getQueue()` 语句声明了一个不传递任何参数、非持久的、唯一的、自动删除的队列，该队列名称由服务器随机产生。

`channel.queueBind(queueName, EXCHANGE_NAME, "")` 为交换器指定队列，并设置了绑定。

3.3.3 例子：Routing

Routing（路由）意味着在消息订阅中可选择性地只订阅部分消息，如图 3-5 所示。比如，在日志系统中，我们可以只接收 `Error` 级别的消息写入文件。同时仍然可以在控制台打印所有日志。

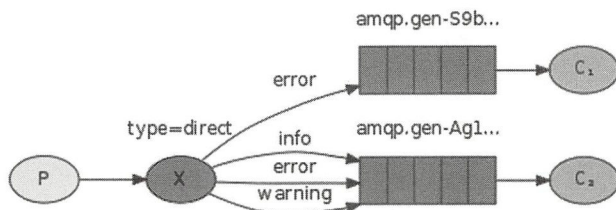


图 3-5 Routing 示意图

在本例中，我们将使用 `direct` 类型的 `exchange`，这样，消息会被推送至 `binding key`（绑定键）和消息发布附带的 `routing key`（路由键）完全匹配的队列。比如：

```
channel.queueBind(queueName, EXCHANGE_NAME, "error");
```

其中的第三个参数 `black` 就是 `routing key`。

1. EmitLogDirect.java 程序

`EmitLogDirect.java` 程序用于日志发送。`EmitLogDirect.java` 代码如下：

```
public class EmitLogDirect {

    private static final Logger LOGGER = LoggerFactory.getLogger(
        EmitLogDirect.class);

    private static final String EXCHANGE_NAME = "direct_logs";

    public static void main(String[] argv) throws Exception {

        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
```



```
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();

channel.exchangeDeclare(EXCHANGE_NAME, "direct");

String severity = getSeverity(argv);
String message = getMessage(argv);

// 为简化程序, 这里的 severity 是 info、warning、error 中的一个
channel.basicPublish(EXCHANGE_NAME, severity, null, message.getBytes(
    "UTF-8"));
LOGGER.info(" [x] Sent '" + severity + "':" + message + "'");

channel.close();
connection.close();
}

private static String getSeverity(String[] strings) {
    if (strings.length < 1)
        return "info";
    return strings[0];
}

private static String getMessage(String[] strings) {
    if (strings.length < 2)
        return "Hello World!";
    return joinStrings(strings, " ", 1);
}

private static String joinStrings(String[] strings, String delimiter,
int startIndex) {
    int length = strings.length;
    if (length == 0)
        return "";
    if (length < startIndex)
        return "";
    StringBuilder words = new StringBuilder(strings[startIndex]);
    for (int i = startIndex + 1; i < length; i++) {
```

```
        words.append(delimiter).append(strings[i]);
    }
    return words.toString();
}
}
```

2. ReceiveLogsDirect.java 程序

ReceiveLogsDirect.java 程序用于日志接收。ReceiveLogsDirect.java 代码如下：

```
public class ReceiveLogsDirect {

    private static final Logger LOGGER = LoggerFactory.getLogger(
        ReceiveLogsDirect.class);

    private static final String EXCHANGE_NAME = "direct_logs";

    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        channel.exchangeDeclare(EXCHANGE_NAME, "direct");
        String queueName = channel.queueDeclare().getQueue();

        if (argv.length < 1) {
            LOGGER.error("Usage: ReceiveLogsDirect [info] [warning] [error]");
            System.exit(1);
        }

        // 为我们感兴趣的 severity 创建一个新的绑定
        for (String severity : argv) {
            channel.queueBind(queueName, EXCHANGE_NAME, severity);
        }

        LOGGER.info(" [*] Waiting for messages. To exit press CTRL+C");

        Consumer consumer = new DefaultConsumer(channel) {
            @Override
```

```

        public void handleDelivery(String consumerTag, Envelope envelope,
        AMQP.BasicProperties properties,
            byte[] body) throws IOException {
            String message = new String(body, "UTF-8");
            LOGGER.info(" [x] Received '" + envelope.getRoutingKey() +
            "' : '" + message + "'");
        }
    };
    channel.basicConsume(queueName, true, consumer);
}
}

```

3. 启动程序

先启动 RabbitMQ 服务器，再使用 Maven 语句 `mvn clean package` 编译程序。编译后，会在 `target` 目录下生成可执行的 `rabbitmq-demo-1.0.0.jar` 文件。

为了演示效果更好，启动了 2 个 `ReceiveLogsDirect`：一个绑定 `error`；另一个绑定 `info`、`warning` 和 `error`。

分别启动 `ReceiveLogsDirect` 执行：

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.ReceiveLogsDirect error
```

以及

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.ReceiveLogsDirect
info warning error
```

启动 `EmitLogDirect` 执行：

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.EmitLogDirect
error "An error will explode."
```

以及

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.EmitLogDirect
info "welcome to waylau.com!"
```

`EmitLogDirect` 发送了 `routing key` 分别为 `error`、`info` 的两条信息，输出如下：

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.EmitLogDirect
error "An error will explode."
```

```
13:41:05.030 [main] INFO com.waylau.rabbitmq.EmitLogDirect - [x] Sent
'error': 'An error will explode.'
```

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.EmitLogDirect
info "welcome to waylau.com!"
13:41:46.258 [main] INFO com.waylau.rabbitmq.EmitLogDirect - [x] Sent
'info':'welcome to waylau.com!'
```

两个 ReceiveLogsDirect 根据自身所绑定的 routing key 收到了不同的消息，分别输出：

```
13:37:38.726 [main] INFO com.waylau.rabbitmq.ReceiveLogsDirect - [*]
Waiting for messages. To exit press CTRL+C
13:41:05.030 [pool-2-thread-4] INFO com.waylau.rabbitmq.ReceiveLogsDirect -
[x] Received 'error':'An error will explode.'
```

以及

```
13:38:08.782 [main] INFO com.waylau.rabbitmq.ReceiveLogsDirect - [*]
Waiting for messages. To exit press CTRL+C
13:41:05.030 [pool-2-thread-4] INFO com.waylau.rabbitmq.ReceiveLogsDirect
- [x] Received 'error':'An error will explode.'
13:41:46.258 [pool-2-thread-5] INFO com.waylau.rabbitmq.ReceiveLogsDirect
- [x] Received 'info':'welcome to waylau.com!'
```

4. 程序详解

在 EmitLogDirect 程序中，channel.basicPublish(EXCHANGE_NAME, severity, null, message.getBytes("UTF-8"))为不同的 routing key 做绑定。

同样，ReceiveLogsDirect 程序只接收 channel.queueBind(queueName, EXCHANGE_NAME, severity)所绑定的 routing key 的消息。

3.3.4 例子：Topics

Topic 类型的 exchange 比 direct 类型拥有更多的灵活性。topic exchange 的 routing key 可以是长度不超过 255 bytes 的字符，以点号“.”进行分割，比如 stock.usd.nyse、nyse.vmw 或 quick.orange.rabbit。需要注意的是，所绑定的 key 必须是相同的格式。其中，有两个特殊的字符：

- * 代表任意一个单词；
- # 代表 0 个或多个单词。

例如，某个 routing key 的格式为<speed>.<colour>.<species>。其中 speed 代表速度，color

代表颜色, species 代表种类。那么 *.orange.* 代表了颜色为 orange 的所有动物, lazy.# 代表了所有懒惰的动物。

Topic 示意图如图 3-6 所示。

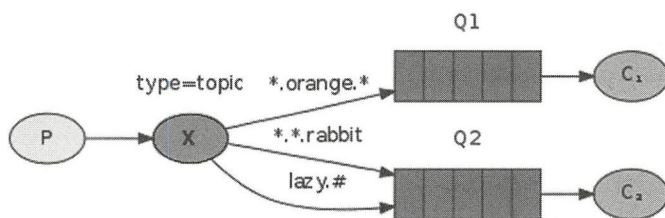


图 3-6 Topic 示意图

1. ReceiveLogsTopic.java 程序

ReceiveLogsTopic.java 程序用于日志接收。ReceiveLogsTopic.java 代码如下：

```

public class ReceiveLogsTopic {

    private static final Logger LOGGER = LoggerFactory.getLogger
(ReceiveLogsTopic.class);

    private static final String EXCHANGE_NAME = "topic_logs";

    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        // 声明一个 topic 类型的 exchange
        channel.exchangeDeclare(EXCHANGE_NAME, "topic");
        String queueName = channel.queueDeclare().getQueue();

        if (argv.length < 1) {
            LOGGER.error("Usage: ReceiveLogsTopic [binding_key]...");
            System.exit(1);
        }

        for (String bindingKey : argv) {
            channel.queueBind(queueName, EXCHANGE_NAME, bindingKey);
        }
    }
}

```

```

        LOGGER.info(" [*] Waiting for messages. To exit press CTRL+C");

        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties,
                byte[] body) throws IOException {
                String message = new String(body, "UTF-8");
                LOGGER.info(" [x] Received '" + envelope.getRoutingKey() +
                    "':" + message + "'");
            }
        };
        channel.basicConsume(queueName, true, consumer);
    }
}

```

2. EmitLogTopic.java 程序

EmitLogTopic.java 程序用于日志发送。EmitLogTopic.java 代码如下：

```

private static final Logger LOGGER = LoggerFactory.getLogger(
    EmitLogTopic.class);

private static final String EXCHANGE_NAME = "topic_logs";

public static void main(String[] argv) {
    Connection connection = null;
    Channel channel = null;
    try {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");

        connection = factory.newConnection();
        channel = connection.createChannel();

        // 声明一个 topic 类型的 exchange
        channel.exchangeDeclare(EXCHANGE_NAME, "topic");

        String routingKey = getRouting(argv);
        String message = getMessage(argv);
    }
}

```

```
        channel.basicPublish(EXCHANGE_NAME, routingKey, null,
message.getBytes("UTF-8"));
        LOGGER.info(" [x] Sent '" + routingKey + "':'" + message + "'");

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (connection != null) {
            try {
                connection.close();
            } catch (Exception ignore) {
            }
        }
    }
}

private static String getRouting(String[] strings) {
    if (strings.length < 1)
        return "anonymous.info";
    return strings[0];
}

private static String getMessage(String[] strings) {
    if (strings.length < 2)
        return "Hello World!";
    return joinStrings(strings, " ", 1);
}

private static String joinStrings(String[] strings, String delimiter,
int startIndex) {
    int length = strings.length;
    if (length == 0)
        return "";
    if (length < startIndex)
        return "";
    StringBuilder words = new StringBuilder(strings[startIndex]);
    for (int i = startIndex + 1; i < length; i++) {
        words.append(delimiter).append(strings[i]);
    }
}
```

```

    }
    return words.toString();
}
}

```

3. 启动程序

先启动 RabbitMQ 服务器，再使用 Maven 语句：`mvn clean package` 编译程序。编译后，会在 `target` 目录下生成可执行的 `rabbitmq-demo-1.0.0.jar` 文件。

为了演示效果，启动了 2 个 `ReceiveLogsTopic`：一个绑定 `kern.*`；另外一个绑定 `kern.*` 和 `*.critical`。

分别启动 `ReceiveLogsTopic` 执行：

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.ReceiveLogsTopic
"kern.*"
```

以及

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.ReceiveLogsTopic
"kern.*" "*.critical"
```

启动 `EmitLogTopic` 执行：

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.EmitLogTopic
"kern.critical" "A critical kernel error"
```

以及

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.EmitLogTopic
"net.critical" "A critical net error"
```

`EmitLogTopic` 输出如下：

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.EmitLogTopic
"kern.critical" "A critical kernel error"
```

```
15:03:03.333 [main] INFO com.waylau.rabbitmq.EmitLogTopic - [x] Sent
'kern.critical': 'A critical kernel error'
```

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.EmitLogTopic
"net.critical" "A critical net error"
```

```
15:03:24.197 [main] INFO com.waylau.rabbitmq.EmitLogTopic - [x] Sent
'net.critical': 'A critical net error'
```


两个 ReceiveLogsTopic 分别输出:

```
15:03:03.333 [pool-2-thread-5] INFO com.waylau.rabbitmq.ReceiveLogsTopic
- [x] Received 'kern.critical': 'A critical kernel error'
```

以及

```
15:03:03.333 [pool-2-thread-5] INFO com.waylau.rabbitmq.ReceiveLogsTopic
- [x] Received 'kern.critical': 'A critical kernel error'
15:03:24.197 [pool-2-thread-6] INFO com.waylau.rabbitmq.ReceiveLogsTopic
- [x] Received 'net.critical': 'A critical net error'
```

4. 程序详解

channel.exchangeDeclare(EXCHANGE_NAME, "topic")声明为 topic 类型的 exchange。

ReceiveLogsTopic 程序只接收 channel.queueBind(queueName, EXCHANGE_NAME, bindingKey) 所绑定的 routing key 的消息。

3.3.5 例子：RPC

本例演示如何在 RabbitMQ 里实现 RPC（远程过程调用）。

这个例子的工作流程如下所示。

- 当客户端启动时，它创建了匿名的单独的回调 queue。
- 客户端实现 RPC 请求时将同时设置两个属性：设置回调 queue 的 replyTo，以及为每个请求设置唯一值 correlationId。
- 请求被发送到一个名为 rpc_queue 的 queue 中。
- RPC worker 或 server 一直在等待那个 queue 的请求。当请求到达时，通过在 replyTo 中指定的 queue 来回复一个消息给客户端。
- 客户端一直等待回调 queue 的数据。当消息到达时，检查 correlationId 属性，如果该属性和它请求的值一致，那么就返回响应结果给程序。

RPC 示意图如图 3-7 所示。

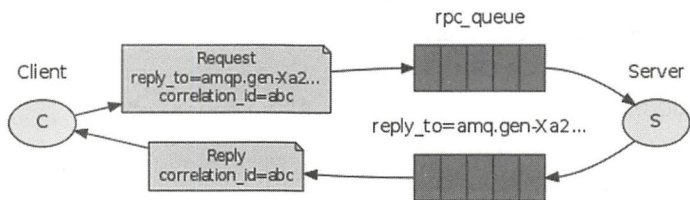


图 3-7 RPC 示意图

1. RPCServer.java 程序

RPCServer.java 是服务器程序。RPCServer.java 代码如下：

```
public class RPCServer {

    private static final Logger LOGGER = LoggerFactory.getLogger
(RPCServer.class);
    private static final String RPC_QUEUE_NAME = "rpc_queue";

    private static int fib(int n) {
        if (n == 0)
            return 0;
        if (n == 1)
            return 1;
        return fib(n - 1) + fib(n - 2);
    }

    public static void main(String[] argv) {
        Connection connection = null;
        Channel channel = null;
        try {
            ConnectionFactory factory = new ConnectionFactory();
            factory.setHost("localhost");

            connection = factory.newConnection();
            channel = connection.createChannel();

            channel.queueDeclare(RPC_QUEUE_NAME, false, false, false, null);

            channel.basicQos(1);

            QueueingConsumer consumer = new QueueingConsumer(channel);
            channel.basicConsume(RPC_QUEUE_NAME, false, consumer);

            LOGGER.info(" [x] Awaiting RPC requests");

            while (true) {
                String response = null;
```

```

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

BasicProperties props = delivery.getProperties();
BasicProperties replyProps = new BasicProperties.Builder()
    .correlationId(props.getCorrelationId())
    .build();

try {
    String message = new String(delivery.getBody(), "UTF-8");
    int n = Integer.parseInt(message);

    LOGGER.info(" [.] fib(" + message + ")");
    response = "" + fib(n);
} catch (Exception e) {
    LOGGER.error(" [.] " + e.toString());
    response = "";
} finally {
    channel.basicPublish("", props.getReplyTo(), replyProps,
response.getBytes("UTF-8"));

    channel.basicAck(delivery.getEnvelope().getDeliveryTag(),
false);
}
} catch (Exception e) {
    e.printStackTrace();
} finally {
    if (connection != null) {
        try {
            connection.close();
        } catch (Exception ignore) {
        }
    }
}
}
}

```

2. RPCClient.java 程序

RPCClient.java 是客户端程序。RPCClient.java 代码如下：

```
public class RPCClient {

    private static final Logger LOGGER = LoggerFactory.getLogger
(RPCClient.class);

    private Connection connection;
    private Channel channel;
    private String requestQueueName = "rpc_queue";
    private String replyQueueName;
    private QueueingConsumer consumer;

    public RPCClient() throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        connection = factory.newConnection();
        channel = connection.createChannel();

        replyQueueName = channel.queueDeclare().getQueue();
        consumer = new QueueingConsumer(channel);
        channel.basicConsume(replyQueueName, true, consumer);
    }

    public String call(String message) throws Exception {
        String response = null;
        String corrId = UUID.randomUUID().toString();

        // 每个请求设置唯一值 correlationId
        BasicProperties props = new BasicProperties.Builder()
.correlationId(corrId).replyTo(replyQueueName).build();

        channel.basicPublish("", requestQueueName, props, message.getBytes
("UTF-8"));

        while (true) {
            QueueingConsumer.Delivery delivery = consumer.nextDelivery();
```




```
// 检查 correlationId 属性是否和它请求的值一致
if (delivery.getProperties().getCorrelationId().equals(corrId)) {
    response = new String(delivery.getBody(), "UTF-8");
    break;
}

return response;
}

public void close() throws Exception {
    connection.close();
}

public static void main(String[] argv) {
    RPCClient fibonacciRpc = null;
    String response = null;
    try {
        fibonacciRpc = new RPCClient();

        LOGGER.info(" [x] Requesting fib(30)");
        response = fibonacciRpc.call("30");
        LOGGER.info(" [.] Got '" + response + "'");
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (fibonacciRpc != null) {
            try {
                fibonacciRpc.close();
            } catch (Exception ignore) {
            }
        }
    }
}
```



3. 启动程序

先启动 RabbitMQ 服务器，再使用 Maven 语句：`mvn clean package` 编译程序。编译后，会在 `target` 目录下生成可执行的 `rabbitmq-demo-1.0.0.jar` 文件。

启动 `RPCServer` 执行如下：

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.RPCServer
```

启动 `RPCClient` 执行如下：

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.RPCClient
```

`RPCServer` 输出如下：

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.RPCServer
16:08:34.197 [main] INFO com.waylau.rabbitmq.RPCServer - [x] Awaiting RPC requests
16:08:46.772 [main] INFO com.waylau.rabbitmq.RPCServer - [.] fib(30)
```

`RPCClient` 输出如下：

```
java -cp target/rabbitmq-demo-1.0.0.jar com.waylau.rabbitmq.RPCClient
16:09:01.966 [main] INFO com.waylau.rabbitmq.RPCClient - [x] Requesting fib(30)
16:09:02.184 [main] INFO com.waylau.rabbitmq.RPCClient - [.] Got '832040'
```

4. 程序详解

通过 `BasicProperties props = new BasicProperties.Builder().correlationId(correlationId).replyTo(replyQueueName).build()` 来给每个请求设置唯一值 `correlationId`。

通过 `delivery.getProperties().getCorrelationId().equals(correlationId)` 来检查 `correlationId` 属性，是否和它请求的值一致，一致了才将返回值交给程序处理。

上面例子中的代码，可以在 <https://github.com/waylau/distributed-systems-technologies-and-cases-analysis> 的 `rabbitmq-demo` 程序中找到。

3.4 Apache RocketMQ

当市面上流行的开源项目无法满足自身业务发展的需要时，企业将不得不走上自研道路。RocketMQ 的出现也是如此。RocketMQ 诞生于阿里巴巴，如今已在全球广泛应用。

2016 年 11 月 21 日，阿里巴巴将 RocketMQ 捐赠给了 Apache 基金会进行孵化。从此，RocketMQ 走出国门，成为更加国际化的一款中间件产品。



1. RocketMQ 物理部署结构

RocketMQ 物理部署结构如图 3-8 所示。

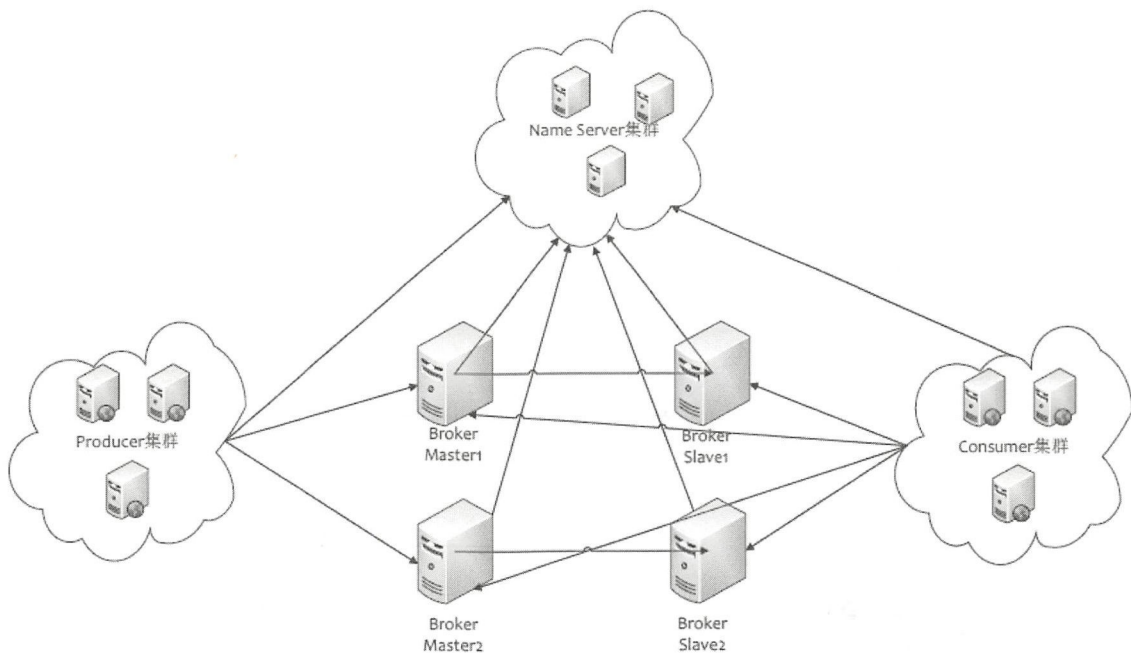


图 3-8 RocketMQ 物理部署结构

RocketMQ 网络部署特点：

- Name Server（名称服务）是一个几乎无状态节点，可集群部署，节点之间无任何信息同步。
- Broker 部署相对复杂，Broker 分为 Master 与 Slave，一个 Master 可以对应多个 Slave，但是一个 Slave 只能对应一个 Master，Master 与 Slave 的对应关系通过指定相同的 BrokerName、不同的 BrokerId 来定义，BrokerId 为 0 表示 Master，非 0 表示 Slave。Master 也可以部署多个。每个 Broker 与 Name Server 集群中的所有节点建立长连接，定时注册 Topic 信息到所有 Name Server。
- Producer 与 Name Server 集群中的其中一个节点（随机选择）建立长连接，定期从 Name Server 取 Topic 路由信息，并向提供 Topic 服务的 Master 建立长连接，且定时向 Master 发送心跳。Producer 完全无状态，可集群部署。
- Consumer 与 Name Server 集群中的一个节点（随机选择）建立长连接，定期从 Name Server 取 Topic 路由信息，并向提供 Topic 服务的 Master、Slave 建立长连接，且定时向 Master、



Slave 发送心跳。Consumer 既可以从 Master 订阅消息，也可以从 Slave 订阅消息，订阅规则由 Broker 配置决定。

2. RocketMQ 逻辑部署结构

RocketMQ 逻辑部署结构如图 3-9 所示。

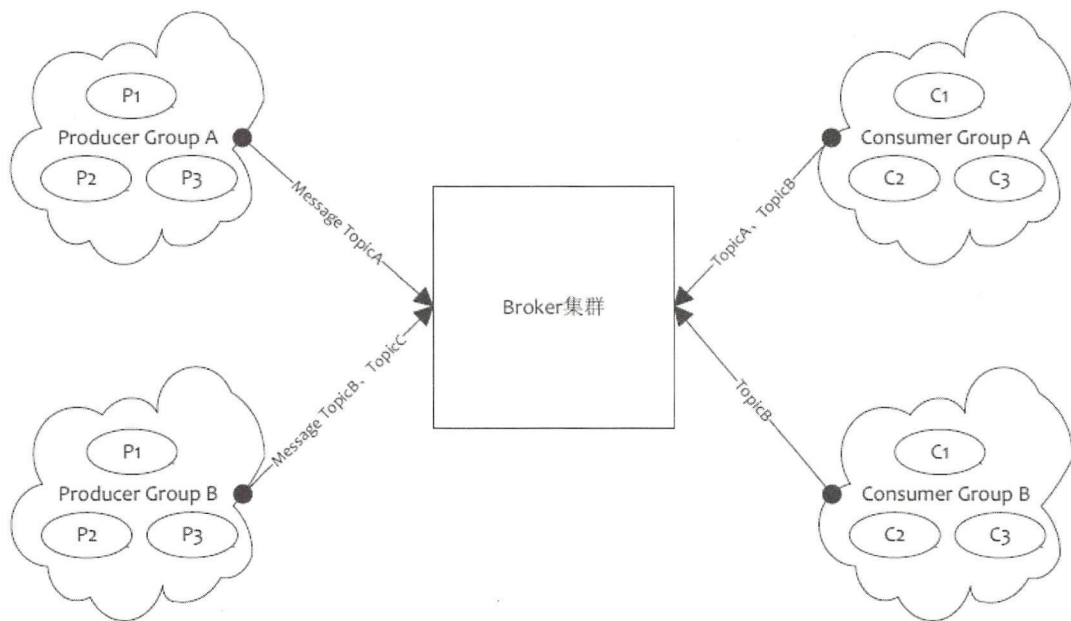


图 3-9 RocketMQ 逻辑部署结构

- **Producer Group:** 用来表示一个发送消息的应用。一个 Producer Group 下包含多个 Producer 实例，可以是多台机器，也可以是一台机器的多个进程，或者是一个进程中的多个 Producer 对象。一个 Producer Group 可以发送多个 Topic 消息，Producer Group 的作用如下：
 - 标识一类 Producer;
 - 可以通过运维工具查询这个发送消息的应用下有多少个 Producer 实例;
 - 发送分布式事务消息时，如果 Producer 中途意外宕机，则 Broker 会主动回调 Producer Group 内的任意一台机器来确认事务状态。
- **Consumer Group:** 用来表示一个消费消息的应用。一个 Consumer Group 下包含多个 Consumer 实例，可以是多台机器，也可以是多个进程，或者是一个进程的多个 Consumer 对象。一个 Consumer Group 下的多个 Consumer 以均摊方式消费信息，如果设置为广播



方式，那么这个 Consumer Group 下的每个实例都消费全量数据。需要注意的是，同一个 Consumer Group 中的 Consumer 实例必须有相同的 topic subscription。可以说，Consumer Group 变相地实现了消息领域最著名的点对点和广播通信模式。

3.4.1 例子：使用 Java 实现 producer-consumer

本例将演示在 producer-consumer（生产者—消费者）模式下，通过 broker 生产和消费消息。

1. Producer.java 程序

Producer 是生产者，用于发送消息。Producer.java 代码如下：

```
public class Producer {

    private static final Logger LOGGER = LoggerFactory.getLogger
(Producer.class);

    private static final String GROUP_NAME = "waylau_com_producer_group";
    private static final String NAME_SERVER = "10.30.22.108:9876";

    public static void main(String[] args) throws MQClientException,
InterruptedException {

        // producerGroup 必须唯一
        DefaultMQProducer producer = new DefaultMQProducer(GROUP_NAME);

        // 设置 name server 地址
        producer.setNamesrvAddr(NAME_SERVER);

        // 在发送消息前，必须调用 start 方法来启动 Producer，只需调用一次即可
        producer.start();

        // 模拟发送 10 条数据
        for (int i = 0; i < 10; i++) {
            try {
                Message msg = new Message(
                    // Message Topic
                    "TopicTest",
                    // Message Tag，对消息进行再归类，方便 Consumer 指定过滤条
                    // 件在 MQ 服务器中过滤
                    "TagA",
```



```

        // Message Body, 对于任何二进制形式的数据, 需要 Producer 与
        // Consumer 协商好一致的序列化和反序列化方式
        ("Welcome to waylau.com! " + i).getBytes());

    // 发送消息
    SendResult sendResult = producer.send(msg);

    LOGGER.info(sendResult.toString());
} catch (Exception e) {
    LOGGER.error(e.getMessage());
    Thread.sleep(1000);
}
}

// 在应用退出前, 可以销毁 Producer 对象
producer.shutdown();
}
}

```

2. Consumer.java 程序

Consumer 是消费者, 用于接收和处理消息。Consumer.java 代码如下:

```

public class Consumer {

    private static final Logger LOGGER = LoggerFactory.getLogger
(Consumer.class);

    private static final String GROUP_NAME = "waylau_com_consumer_group";
    private static final String NAME_SERVER = "10.30.22.108:9876";

    public static void main(String[] args) throws InterruptedException,
MQClientException {

        // consumerGroup 必须唯一
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer(GROUP_NAME);

        // 设置 Consumer 第一次启动时从队列头部还是从队列尾部开始消费
        // 如果非第一次启动, 那么按照上次消费的位置继续消费
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_
OFFSET);
    }
}

```



```
// 设置 name server 地址
consumer.setNamesrvAddr (NAME_SERVER);

// 订阅 Topic
consumer.subscribe("TopicTest", "*");

// 监听消息
consumer.registerMessageListener(new MessageListenerConcurrently() {

    @Override
    public ConsumeConcurrentlyStatus consumeMessage(
        List<MessageExt> msgs, ConsumeConcurrentlyContext context) {
        LOGGER.info(Thread.currentThread().getName() + " Receive New
Messages: " + msgs);
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
});

consumer.start();

LOGGER.info("Consumer Started.");
}
```

3. 启动程序

先启动 RocketMQ 服务器，再使用 Maven 语句：mvn clean package 编译程序。编译后，会在 target 目录下生成可执行的 rocketmq-demo-1.0.0.jar 文件。

启动 Consumer 执行如下：

```
java -cp target/rocketmq-demo-1.0.0.jar com.waylau.rocketmq.Consumer
```

启动 Producer 执行如下：

```
java -cp target/rocketmq-demo-1.0.0.jar com.waylau.rocketmq.Producer
```

Producer 的输出如下：

```
12:09:23.885 [main] INFO RocketmqRemoting - createChannel: connect remote
host[10.30.22.108:10909] success, DefaultChannelPromise@a4102b8(success)
12:09:23.931 [main] INFO com.waylau.rocketmq.Producer - SendResult
[sendStatus=SEND_OK, msgId=0A1E166C14F85C647E0561E16EA80000,offsetMsgId=
```



```
0A1E166C00002A9F0000000000000E88, messageQueue=MessageQueue [topic=TopicTest,
brokerName=admin-PC, queueId=3], queueOffset=5]
12:09:23.939 [main] INFO com.waylau.rocketmq.Producer - SendResult
[sendStatus=SEND_OK, msgId=0A1E166C14F85C647E0561E16EDB0001,offsetMsgId=
0A1E166C00002A9F0000000000000F42, messageQueue=MessageQueue [topic=TopicTest,
brokerName=admin-PC, queueId=0], queueOffset=5]
12:09:23.947 [main] INFO com.waylau.rocketmq.Producer - SendResult
[sendStatus=SEND_OK, msgId=0A1E166C14F85C647E0561E16EE40002,offsetMsgId=
0A1E166C00002A9F0000000000000FFC, messageQueue=MessageQueue [topic=TopicTest,
brokerName=admin-PC, queueId=1], queueOffset=5]
12:09:23.953 [main] INFO com.waylau.rocketmq.Producer - SendResult
[sendStatus=SEND_OK, msgId=0A1E166C14F85C647E0561E16EEC0003,offsetMsgId=
0A1E166C00002A9F000000000000010B6, messageQueue=MessageQueue [topic=TopicTest,
brokerName=admin-PC, queueId=2], queueOffset=5]
12:09:23.959 [main] INFO com.waylau.rocketmq.Producer - SendResult
[sendStatus=SEND_OK, msgId=0A1E166C14F85C647E0561E16EF20004,offsetMsgId=
0A1E166C00002A9F00000000000001170, messageQueue=MessageQueue [topic=TopicTest,
brokerName=admin-PC, queueId=3], queueOffset=6]
12:09:23.966 [main] INFO com.waylau.rocketmq.Producer - SendResult
[sendStatus=SEND_OK, msgId=0A1E166C14F85C647E0561E16EF80005,offsetMsgId=
0A1E166C00002A9F0000000000000122A, messageQueue=MessageQueue [topic=TopicTest,
brokerName=admin-PC, queueId=0], queueOffset=6]
12:09:23.981 [main] INFO com.waylau.rocketmq.Producer - SendResult
[sendStatus=SEND_OK, msgId=0A1E166C14F85C647E0561E16EFF0006,offsetMsgId=
0A1E166C00002A9F000000000000012E4, messageQueue=MessageQueue [topic=TopicTest,
brokerName=admin-PC, queueId=1], queueOffset=6]
12:09:24.000 [main] INFO com.waylau.rocketmq.Producer - SendResult
[sendStatus=SEND_OK, msgId=0A1E166C14F85C647E0561E16F140007,offsetMsgId=
0A1E166C00002A9F0000000000000139E, messageQueue=MessageQueue [topic=TopicTest,
brokerName=admin-PC, queueId=2], queueOffset=6]
12:09:24.010 [main] INFO com.waylau.rocketmq.Producer - SendResult
[sendStatus=SEND_OK, msgId=0A1E166C14F85C647E0561E16F210008,offsetMsgId=
0A1E166C00002A9F00000000000001458, messageQueue=MessageQueue [topic=TopicTest,
brokerName=admin-PC, queueId=3], queueOffset=7]
12:09:24.019 [main] INFO com.waylau.rocketmq.Producer - SendResult
[sendStatus=SEND_OK, msgId=0A1E166C14F85C647E0561E16F2A0009,offsetMsgId=
0A1E166C00002A9F00000000000001512, messageQueue=MessageQueue [topic=TopicTest,
brokerName=admin-PC, queueId=0], queueOffset=7]
```





Consumer 的输出如下:

```
14:43:58.428 [ConsumeMessageThread_1] INFO com.waylau.rocketmq.Consumer -
ConsumeMessageThread_1 Receive New Messages: Welcome to waylau.com! 0
14:43:58.433 [ConsumeMessageThread_3] INFO com.waylau.rocketmq.Consumer -
ConsumeMessageThread_3 Receive New Messages: Welcome to waylau.com! 3
14:43:58.441 [ConsumeMessageThread_2] INFO com.waylau.rocketmq.Consumer -
ConsumeMessageThread_2 Receive New Messages: Welcome to waylau.com! 1
14:43:58.445 [ConsumeMessageThread_4] INFO com.waylau.rocketmq.Consumer -
ConsumeMessageThread_4 Receive New Messages: Welcome to waylau.com! 2
14:43:58.446 [ConsumeMessageThread_5] INFO com.waylau.rocketmq.Consumer -
ConsumeMessageThread_5 Receive New Messages: Welcome to waylau.com! 4
14:43:58.453 [ConsumeMessageThread_6] INFO com.waylau.rocketmq.Consumer -
ConsumeMessageThread_6 Receive New Messages: Welcome to waylau.com! 5
14:43:58.458 [ConsumeMessageThread_7] INFO com.waylau.rocketmq.Consumer -
ConsumeMessageThread_7 Receive New Messages: Welcome to waylau.com! 6
14:43:58.467 [ConsumeMessageThread_8] INFO com.waylau.rocketmq.Consumer -
ConsumeMessageThread_8 Receive New Messages: Welcome to waylau.com! 7
14:43:58.476 [ConsumeMessageThread_9] INFO com.waylau.rocketmq.Consumer -
ConsumeMessageThread_9 Receive New Messages: Welcome to waylau.com! 8
14:43:58.482 [ConsumeMessageThread_10] INFO com.waylau.rocketmq.Consumer -
ConsumeMessageThread_10 Receive New Messages: Welcome to waylau.com!9
```

4. 程序详解

通过 `consumer.setNamesrvAddr(NAME_SERVER)` 及 `producer.setNamesrvAddr(NAME_SERVER)` 来设置 name server 的地址, 该地址应与服务器启动时指定的地址一致。

通过 `consumer.subscribe("TopicTest", "*")` 订阅感兴趣的 topic 和 tag, 本例的 tag 为 "*", 表示对任意 tag 都感兴趣。如果对某几个 tag 感兴趣, 则可以设置为 "TagA || TagC || TagD"。

生产者 Producer 所发送的 Message 主要参数为 (String topic, String tags, byte[] body), 分别指定了 topic、tag 及消息体的内容字节。

上面例子的代码可以在 <https://github.com/waylau/distributed-systems-technologies-and-cases-analysis> 的 rocketmq-demo 程序中找到。

3.4.2 RocketMQ 最佳实践

本节内容是开发者在 RocketMQ 实际应用中对所遇到的问题进行的思考和测试, 以及最终采用的最佳解决方案。下面部分内容节选自 RocketMQ 开发团队所著的《RocketMQ 开发指南》。



1. Producer 最佳实践

(1) 一个应用尽可能用一个 Topic，消息子类型用 tags 来标识，tags 可以由应用自由设置。只有发送消息设置了 tags，消费方在订阅消息时才可以利用 tags 在 broker 中做消息过滤。例如：

```
message.setTags("TagA");
```

(2) 每个消息在业务层面的唯一标识码要设置到 keys 字段，方便将来定位消息丢失问题。由于是哈希索引，请务必保证 key 尽可能唯一，这样可以避免潜在的哈希冲突。例如：

```
String orderId = "20160820083112001";  
message.setKeys(orderId);
```

(3) 消息发送成功或失败，要打印消息日志，务必要打印 sendresult 和 key 字段。

(4) 发送消息方法，只要不抛异常，就代表发送成功。但是发送成功会有多个状态，sendresult 定义了 SendStatus 枚举类型：

- SEND_OK——消息发送成功；
- FLUSH_DISK_TIMEOUT——消息发送成功，但是服务器刷盘（“刷盘”是指把数据从内存写到硬盘的过程）超时，消息已经进入服务器队列，只有此时服务器宕机，消息才会丢失；
- FLUSH_SLAVE_TIMEOUT——消息发送成功，但是服务器同步至 Slave 时超时，消息已经进入服务器队列，只有此时服务器宕机，消息才会丢失；
- SLAVE_NOT_AVAILABLE——消息发送成功，但此时 Slave 不可用，消息已经进入服务器队列，只有此时服务器宕机，消息才会丢失。

对于强调发送消息顺序的应用，由于顺序消息的局限性，可能会涉及主备自动切换的问题，所以如果 sendresult 的状态不等于 SEND_OK，则应该尝试重试，对于其他类型的应用，则没有必要这样。

(5) 对于消息不可丢失的应用，务必要有消息重发机制。例如，消息发送失败，存储到数据库，能有定时程序尝试重发或人工触发重发。

Producer 的 send 方法本身支持内部重试，重试逻辑如下：

- 最多重试 3 次；
- 如果发送失败，则轮转到下一个 Broker；
- 方法的总耗时时间不超过 sendMsgTimeout 设置的值，默认为 10s。所以，如果本身向 Broker 发送消息产生超时异常，则不会重试。

以上策略仍然不能保证消息一定发送成功，为了保证消息一定成功，建议应用这样做：如



果调用 `send` 同步方法发送失败，则尝试将消息存储到数据库，由后台线程定时重试，保证消息一定到达 Broker。

上述数据库重试方式没有集成到 MQ 客户端内部实现，而是要求应用自己去完成，主要基于以下几点考虑：

- MQ 的客户端设计为无状态模式，方便任意的水平扩展，且对机器资源的消耗仅仅是 CPU、内存和网络；
- 如果 MQ 客户端内部集成一个 KV 存储模块，那么数据只有同步落盘才能较可靠，而同步落盘本身的性能开销较大，所以通常会采用异步落盘；又因为应用关闭过程不受 MQ 运维人员控制，可能经常会发生 kill-9 这样的暴力方式的关闭，造成数据没有及时落盘而丢失；
- Producer 所在机器的可靠性较低，一般为虚拟机，不适合存储重要数据。

综上，建议重试过程交由应用来控制。

（6）某些应用如果不关注消息是否发送成功，则直接使用 `sendOneWay` 方法发送消息。

例如，在 RPC 调用应用中，通常是这样的一个过程：

- 客户端发送请求到服务器；
- 服务器处理该请求；
- 服务器返回响应给客户端。

所以一个 RPC 的耗时时间是上述三个步骤的总和，而某些场景要求耗时非常短，但是对可靠性要求并不高。例如，日志收集类应用，此类应用可以采用 `sendOneWay` 形式调用。`sendOneWay` 形式只发送请求不等待应答，而发送请求在客户端实现层面仅仅是一个操作系统调用的开销，即将数据写入客户端的 socket 缓冲区，此过程耗时通常在微秒级。

2. Consumer 最佳实践

（1）消费过程要做到幂等（即消费端去重）。

RocketMQ 无法避免消息重复，所以如果业务对消费重复非常敏感，则务必要在业务局面去重，有以下几种去重方式：

- 将消息的唯一键，可以是 `msgId`，也可以是消息内容中的唯一标识字段，例如，订单 Id 等，消费之前判断是否在数据库或 Tair（全局 KV 存储）中存在，如果不存在则插入，并消费，否则跳过（实际过程要考虑原子性问题，判断是否存在可以尝试插入，如果报主键冲突，则插入失败，直接跳过）。`msgId` 一定是全局唯一标识符，但是可能会存在同样的消息有两个不同 `msgId` 的情况（有多种原因），这种情况可能会使业务上重复消



费，建议最好使用消息内容中的唯一标识字段去重。

- 使用业务局面的状态机去重。

(2) 尽量使用批量消费方式，可以在很大程度上提高消费吞吐量。

提高消费并行度

绝大部分消息消费行为属于 I/O 密集型，即可能是操作数据库，或者调用 RPC，消费速度在于后端数据库或外部系统的吞吐量，通过增加消费并行度，可以提高总的消费吞吐量，但是并行度增加到一定程度，消费吞吐量反而会下降，如图 3-10 所示，呈现抛物线形式。所以应用必须要设置合理的并行度。CPU 密集型应用除外。

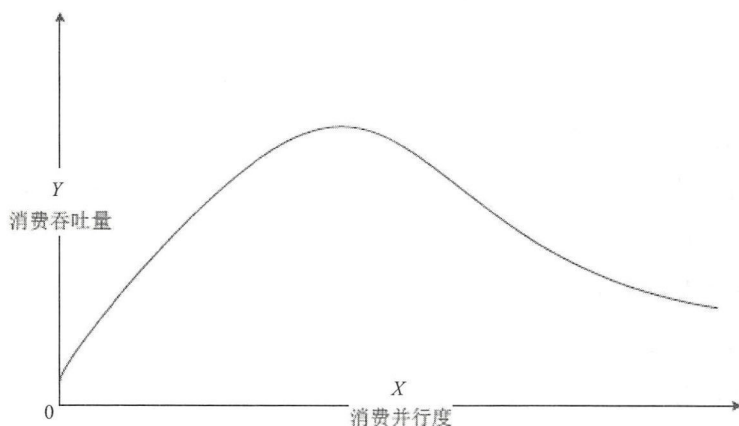


图 3-10 消费并行度与消费吞吐量的关系

修改消费并行度方法

- 同一个 ConsumerGroup 下，通过增加 Consumer 实例数量来提高并行度，超过订阅队列数的 Consumer 实例无效。可以通过加机器，或者在已有机器启动多个进程的方式来实现。
- 提高单个 Consumer 的消费并行线程，通过修改以下参数来实现：

```
consumeThreadMin  
consumeThreadMax
```

批量方式消费

某些业务流程如果支持批量消费，则可以在很大程度上提高消费吞吐量，例如，订单扣款类应用，一次处理一个订单耗时 1 秒，一次处理 10 个订单可能只耗时 2 秒，这样即可大幅度提高消费的吞吐量，通过设置 consumer 的 consumeMessageBatchMaxSize 返回参数，默认是 1，即一次只消费一条消息，如设置为 N，那么每次消费的消息数小于等于 N。



跳过非重要消息

发生消息堆积时，如果消费速度一直追不上发送速度，则可以选择丢弃不重要的消息。如何判断消费发生了堆积呢？

```
public ConsumeConcurrentlyStatus consumeMessage(
    List<MessageExt> msgs, ConsumeConcurrentlyContext context) {
    LOGGER.info(Thread.currentThread().getName() + " Receive New Messages:
" + msgs);

    // 跳过非重要消息。当某个队列的消息数堆积到 100000 条以上时，
    // 则尝试丢弃部分或全部消息，这样就可以快速追上发送消息的速度
    long offset = msgs.get(0).getQueueOffset();
    String maxOffset = msgs.get(0).getProperty(MessageConst.PROPERTY_MAX_
OFFSET);
    long diff = Long.parseLong(maxOffset) - offset;
    if (diff > 100000) {

        // TODO 消息堆积情况的特殊处理
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
}
```

如以上代码所示，当某个队列的消息数堆积到 100000 条以上时，则尝试丢弃部分或全部消息，这样就可以快速追上发送消息的速度。

(3) 优化每条消息消费过程。

某条消息的消费过程如下：

- 根据消息从数据库查询数据 1；
- 根据消息从数据库查询数据 2；
- 复杂的业务计算；
- 向数据库插入数据 3；
- 向数据库插入数据 4。

这条消息的消费过程与数据库交互了 4 次，如果按照每次耗时 5ms 计算，那么总耗时为 20ms，假设业务计算耗时 5ms，那么总耗时为 25ms，如果能把 4 次 DB 交互优化为 2 次，那么总耗时就可以优化到 15ms，也就是说，总体性能提高了 40%。



对于 MySQL 等数据库，如果部署在磁盘中，那么与数据库进行交互，如果数据没有命中 cache，则每次交互的 RT 会直线上升，如果采用 SSD，则 RT 上升趋势要明显好于磁盘。

个别应用可能会遇到这种情况：在线下压测消费过程中，DB 表现得非常好，每次 RT 都很短，但是上线运行一段时间，RT 就会变长，消费吞吐量直线下降。主要原因是线下压测时间过短，线上运行一段时间后，cache 命中率下降，那么 RT 就会增加。建议在线下压测时，要测试足够长时间，尽可能模拟线上环境。在压测过程中，数据的分布也很重要，数据不同，可能 cache 的命中率也会完全不同。

（4）打印消费日志。

如果消息量较少，则建议在消费入口方法中打印消息，方便排查问题。

```
public ConsumeConcurrentlyStatus consumeMessage(  
    List<MessageExt> msgs, ConsumeConcurrentlyContext context) {  
  
    // 打印日志  
    LOGGER.info(Thread.currentThread().getName() + " Receive New Messages:  
" + msgs);  
  
    // TODO 消息处理逻辑  
  
    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;  
}
```

如果能打印每条消息的消费耗时，那么在排查消费慢等线上问题时会更方便。

（5）利用服务器过滤消息，避免多余的传输。

3.5 Apache Kafka

Apache Kafka 是一种高吞吐量的分布式发布—订阅消息系统，可以提供消息的持久化，即使 TB 量级的消息存储也能够保持长时间的稳定性能。同时，Kafka 也支持 Hadoop 并行数据加载。

Kafka 生产者—消费者逻辑部署结构如图 3-11 所示。

服务端和客户端的通信是常见的 TCP 协议。Kafka 支持多种语言的客户端，包括 Java、C/C++、Python、Go（AKA Golang）、Erlang、.NET、Clojure、Ruby、Node.js、Proxy、Perl、stdin/stdout、PHP、Rust、Storm、Scala DSL、Clojure 等。



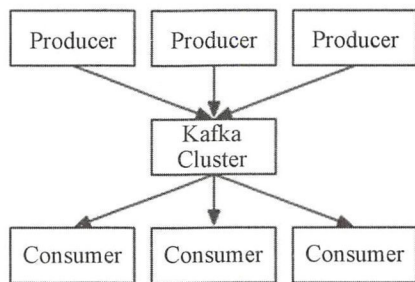


图 3-11 Kafka 生产者—消费者逻辑部署结构

Kafka 是一个基于分布式的消息发布—订阅系统，它被设计成快速、可扩展、持久的。与其他消息发布—订阅系统类似，Kafka 在主题中保存消息的信息。生产者向主题写入数据，消费者从主题读取数据。由于 Kafka 的特性是支持分布式的，也是基于分布式的，所以主题可以在多个节点上被分区和覆盖。

消息是一个字节数组，程序员可以在这些字节数组中存储任何对象，支持的数据格式包括 String、JSON、Avro。Kafka 通过给每一个消息绑定一个键值的方式来保证生产者可以把所有的消息发送到指定位置。属于某一个消费者群组的消费者订阅了一个主题，通过该订阅消费者可以跨节点接收所有与该主题相关的消息，每一个消息只会发送给群组中的一个消费者，所有拥有相同键值的消息都会被确保发给这一个消费者。

Kafka 的设计中将每一个主题分区当作一个具有顺序排列的日志。处于同一个分区中的消息都被设置了一个唯一的偏移量。Kafka 只会跟踪未读消息，一旦消息被置为已读状态，Kafka 就不会再去管理它了。Kafka 的生产者负责在消息队列中对生产出来的消息保证一定时间的占有，消费者负责追踪每一个主题（可以理解为一个日志通道）的消息并及时获取它们。基于这样的设计，Kafka 可以在消息队列中保存大量开销很小的数据，并且支持大量的消费者订阅。

3.5.1 Apache Kafka 的核心概念

1. Topic 和日志

先来看一下 Kafka 提供的一个抽象概念：Topic。

一个 Topic 是对一组消息的归纳。Kafka 对每个 Topic 的日志进行了分区，如图 3-12 所示。

每个分区都由一系列有序的、不可变的消息组成，这些消息被连续地追加到分区中。分区中的每个消息都有一个连续的叫作 offset 的序列号，用在分区中唯一地标识这个消息。



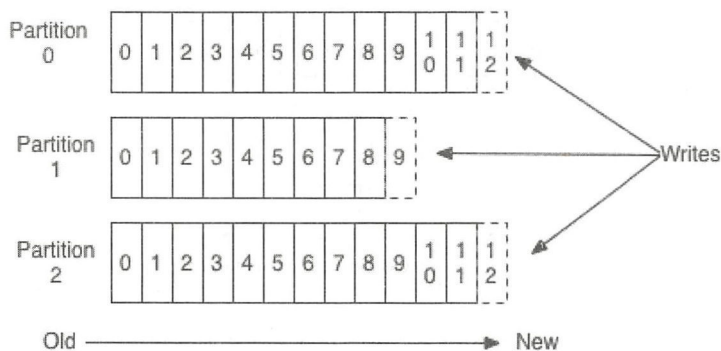


图 3-12 Kafka Topic 分区示意图

在一个可配置的时间段内，Kafka 集群保留所有发布的消息，不管这些消息有没有被消费。比如，如果消息的保存策略被设置为 2 天，那么在一个消息被发布的两天时间内，它都是可以被消费的。之后它将被丢弃以释放空间。Kafka 的性能是和数据量无关的常量级的，所以保留大量的数据并不是问题。

实际上每个 Consumer 唯一需要维护的数据是消息在日志中的位置，也就是“offset”。这个 offset 由 Consumer 来维护：一般情况下随着 Consumer 不断地读取消息，这个 offset 的值不断增加，但其实 Consumer 可以以任意的顺序读取消息，比如它可以将 offset 设置成为一个旧的值来重读之前的消息。

以上特点的结合，使 Kafka Consumer 非常轻量级：它们可以在不对集群和其他 Consumer 造成影响的情况下读取消息。我们可以使用命令“tail”来读取消息而不会对其他正在消费消息的 Consumer 造成影响。

将日志分区可以达到以下目的：每个日志的数量不会太大，可以在单个服务上保存。另外每个分区可以单独发布和消费，为并发操作 Topic 提供了可能。

2. 分布式

每个分区在 Kafka 集群的若干服务中都有副本，这样，这些持有副本的服务可以共同处理数据和请求，副本数量是可以配置的。副本使 Kafka 具备了容错能力。

每个分区都由一个服务器作为“leader”，零或若干服务器作为“follower”。leader 负责处理消息的读和写，follower 则去复制 leader。如果 leader 宕机了，则 follower 中的一台服务器会自动成为 leader。集群中的每个服务都会扮演两个角色：作为它所持有的一部分分区的 leader，同时作为其他分区的 follower，这样集群就会有较好的负载均衡。



3. Producer

Producer 将消息发布到它指定的 Topic 中, 并负责决定发布到哪个分区。通常简单地由负载均衡机制随机选择分区, 但也可以通过特定的分区函数选择分区。一般来说, 常使用的是第二种。

4. Consumer

发布消息通常有两种模式: queue (队列) 模式和 Publish/Subscribe (发布—订阅) 模式。在 queue 模式中, Consumer 可以同时从服务端读取消息, 每个消息只被其中一个 Consumer 读取; 在 Publish/Subscribe 模式中, 消息被广播到所有的 Consumer 中。Consumer 可以加入一个 Consumer Group, 共同竞争一个 Topic, Topic 中的消息将被分发到 Consumer Group 中的一个成员中。

同一 Group 中的 Consumer 可以在不同的程序中, 也可以在不同的机器上。

如果所有的 Consumer 都在一个 Consumer Group 中, 则为传统的 queue 模式, 在各 Consumer 中实现负载均衡。

如果所有的 Consumer 都不在不同的 Consumer Group 中, 则为 Publish/Subscribe 模式, 消息都被分发到所有的 Consumer 中。

更常见的是, 每个 Topic 都有若干数量的 Consumer Group, 每个 Consumer Group 都是一个逻辑上的“订阅者”, 为了容错和更好的稳定性, 每个 Consumer Group 由若干 Consumer 组成。这其实就是一个 Publish/Subscribe 模式, 只不过订阅者是 Consumer Group 而不是单个 Consumer。

Consumer Group 示意图如图 3-13 所示。

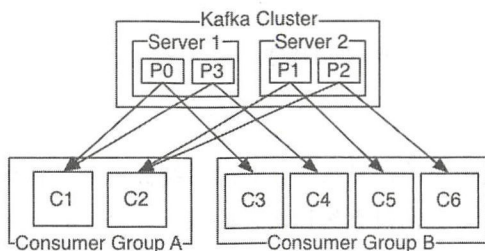


图 3-13 Consumer Group 示意图

相比传统的消息系统, Kafka 可以很好地保证有序性。

传统的队列在服务器上保存有序的消息, 如果多个 Consumer 同时从这个服务器消费消息, 则服务器会以消息存储的顺序向 Consumer 分发消息。虽然服务器按顺序发布消息, 但是消息是被异步地分发到各个 Consumer 上的, 所以当消息到达时可能已经失去了原来的顺序, 这意味着并发消费将导致顺序错乱。为了避免故障, 这样的消息系统通常使用“专用 Consumer”的



概念，其实就是只允许一个消费者消费消息，当然这就意味着失去了并发性。

在这方面 Kafka 做得更好。通过分区概念，Kafka 可以在多个 Consumer 进程并发的情况下提供较好的有序性和负载均衡。将每个分区只分发给一个 Consumer Group，这样一个分区就只被这个 Group 的一个 Consumer 消费，就可以顺序地消费这个分区的信息。因为有多分区，依然可以在多个 Consumer Group 之间进行负载均衡。注意 Consumer Group 的数量不能多于分区的数量，也就是有多少分区就允许多少并发消费。

Kafka 只能保证一个分区之内消息的有序性，在不同的分区之间是不可行的，但即使这样，也已经可以满足大部分应用的需求。如果需要满足 Topic 中所有消息的有序性，那就让这个 Topic 只有一个分区，当然也就只有一个 Consumer Group 消费它。

5. 保障

在一个高层次的 Kafka 中提供了以下保障：

- 由 Producer 发送到一个特定的 Topic 分区的信息将按它们被添加时的顺序来发送。也就是说，如果由相同的 Producer 来发送两个消息 M1、M2，若 M1 先被发送，那么在日志中 M1 将比 M2 具有较低的 offset。
- Consumer 实例以日志中存储的顺序来查看消息。
- 对于复制因子为 N 的 Topic，将能承受最多 N-1 服务器的故障，而不会丢失提交到日志的任何消息。

3.5.2 Apache Kafka 的使用场景

下面是 Apache Kafka 常见的一些使用场景。

1. 消息

Kafka 可以更好地替换传统的消息系统。消息系统被用于各种场景（解耦数据生产者、缓存未处理的消息等），与大多数消息系统比较，Kafka 有更好的吞吐量、内置分区、副本和故障转移，这有利于处理大规模的消息。

消息往往用于较低的吞吐量，但需要低的端到端延迟，并需要提供强大的耐用性的保证。而在这一领域的 Kafka 对比于传统的消息系统毫不逊色。

2. 网站活动追踪

Kafka 设计最初就是用于用户的活动追踪，将网站的活动（网页浏览、搜索或其他用户的操作信息）发布到不同的话题中心，这些消息可实时处理、实时监测，也可加载到 Hadoop 或

离线处理数据仓库。

3. 指标

Kafka 也常用于监测数据，将分布式应用程序生成的统计数据集中聚合。

4. 日志聚合

使用 Kafka 来代替一个日志聚合的解决方案。

5. 流处理

Kafka 消息处理包含多个阶段。其中原始输入数据是从 Kafka 主题开始消费的，然后汇总，或者以其他的方式处理转化为新主题。例如，一个推荐新闻文章，文章内容可能从“articles”主题获取，然后进一步处理内容，得到处理后的新内容，最后推荐给用户。这种处理是基于单个主题的实时数据流。从 0.10.0.0 版本开始，轻量但功能强大的流处理就能进行这样的数据处理了。

除了 Kafka Streams，还有 Apache Storm 和 Apache Samza 可用来进行流处理。

6. 事件采集

事件采集是一种应用程序的设计风格，其中，状态的变化可以根据时间的顺序记录下来，Kafka 支持这种存储日志数据非常大的场景。

7. 提交日志

Kafka 可以作为一种分布式的外部提交日志，它可以在节点之间复制数据，并且可以帮助失败的节点恢复数据使其重新同步。Kafka 的日志压缩功能很好地支持了这种用法，这种用法类似于 Apache BookKeeper 项目。

3.6 实战：基于 JMS 的消息发送和接收

下面我们将基于 JMS 来实现消息的发送和接收。

3.6.1 项目概述

我们将创建一个名为“jms-msg”的应用。在该应用中，我们模拟了生成者、消费者、队列和订阅等用法。

为了能够正常运行该应用，需要在应用中添加如下依赖：

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <spring.version>5.0.5.RELEASE</spring.version>
  <junit.version>4.12</junit.version>
  <activemq.version>5.15.3</activemq.version>
</properties>
```

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-all</artifactId>
    <version>${activemq.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jms</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
```



```

        <artifactId>spring-test</artifactId>
        <version>${spring.version}</version>
    </dependency>
</dependencies>

```

其中，我们使用 Apache ActiveMQ 实现了 JMS，并集成了 Spring。

3.6.2 项目配置

以下是 Spring 基于 XML 的核心配置内容：

```

<!-- 配置 JMS 连接工厂 -->
<bean id="connectionFactory"
    class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="failover:(tcp://localhost:61616)" />
</bean>

<!-- 定义消息队列 (Queue) -->
<bean id="queueDestination"
    class="org.apache.activemq.command.ActiveMQQueue">
    <!-- 设置消息队列的名字 -->
    <constructor-arg>
        <value>queue1</value>
    </constructor-arg>
</bean>

<!-- 配置 JMS 模板 (Queue)，Spring 提供了 JMS 工具类，它负责发送、接收消息。 -->
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="defaultDestination" ref="queueDestination" />
    <property name="receiveTimeout" value="10000" />
</bean>

<!-- queue 消息生产者 -->
<bean id="producerService"
    class="com.waylau.spring.jms.queue.ProducerServiceImpl">
    <property name="jmsTemplate" ref="jmsTemplate"></property>
</bean>

```

```
<!--queue 消息消费者 -->
<bean id="consumerService"
      class="com.waylau.spring.jms.queue.ConsumerServiceImpl">
    <property name="jmsTemplate" ref="jmsTemplate"></property>
</bean>

<!-- 定义消息队列 (Queue) -->
<bean id="queueDestination2"
      class="org.apache.activemq.command.ActiveMQQueue">
    <!-- 设置消息队列的名字 -->
    <constructor-arg>
        <value>queue2</value>
    </constructor-arg>
</bean>

<!-- 消息队列监听者 (Queue) -->
<bean id="queueMessageListener"
      class="com.waylau.spring.jms.queue.QueueMessageListener" />

<!-- 消息监听容器 (Queue) -->
<bean id="jmsContainer"
      class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destination" ref="queueDestination2" />
    <property name="messageListener" ref="queueMessageListener" />
</bean>

<!-- 定义消息主题 (Topic) -->
<bean id="topicDestination" class="org.apache.activemq.command.ActiveMQTopic">
    <constructor-arg>
        <value>guo_topic</value>
    </constructor-arg>
</bean>

<!-- 配置 JMS 模板 (Topic) -->
<bean id="topicJmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="defaultDestination" ref="topicDestination" />
```

```
<property name="pubSubDomain" value="true" />
<property name="receiveTimeout" value="10000" />
</bean>

<!--Topic 消息发布者 -->
<bean id="topicProvider" class="com.waylau.spring.jms.topic.TopicProvider">
    <property name="topicJmsTemplate" ref="topicJmsTemplate"></property>
</bean>

<!-- 消息主题监听者 (Topic) -->
<bean id="topicMessageListener"
    class="com.waylau.spring.jms.topic.TopicMessageListener" />

<!-- 消息主题监听者 (Topic) 2 -->
<bean id="topicMessageListener2"
    class="com.waylau.spring.jms.topic.TopicMessageListener2" />

<!-- 主题监听容器 (Topic) -->
<bean id="topicJmsContainer"
    class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destination" ref="topicDestination" />
    <property name="messageListener" ref="topicMessageListener" />
</bean>

<!-- 主题监听容器 (Topic) 2 -->
<bean id="topicJmsContainer2"
    class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destination" ref="topicDestination" />
    <property name="messageListener" ref="topicMessageListener2" />
</bean>

<!--这是 sessionAwareQueue 的目的地 -->
<bean id="sessionAwareQueue" class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg>
        <value>sessionAwareQueue</value>
    </constructor-arg>
```

```
</bean>

<!-- 可以获取 session 的 MessageListener -->
<bean id="consumerSessionAwareMessageListener"
      class="com.waylau.spring.jms.queue.ConsumerSessionAwareMessageListener">
    <property name="destination" ref="queueDestination" />
</bean>

<!-- 监听 sessionAwareQueue 队列的消息，把回复消息写入 queueDestination，再指向
队列，即 queue1 -->
<bean id="sessionAwareListenerContainer"
      class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destination" ref="sessionAwareQueue" />
    <property name="messageListener" ref="consumerSessionAwareMessageListener" />
</bean>

<!--这是 adapterQueue 的目的地 -->
<bean id="adapterQueue" class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg>
        <value>adapterQueue</value>
    </constructor-arg>
</bean>

<!-- 消息监听适配器 -->
<bean id="messageListenerAdapter"
      class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <property name="delegate">
        <bean class="com.waylau.spring.jms.queue.ConsumerListener" />
    </property>
    <property name="defaultListenerMethod" value="receiveMessage" />
</bean>

<!-- 消息监听适配器对应的监听容器 -->
<bean id="messageListenerAdapterContainer"
      class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destination" ref="adapterQueue" />
```



```

<!-- 使用 MessageListenerAdapter 作为消息侦听器 -->
<property name="messageListener" ref="messageListenerAdapter" />
</bean>

```

3.6.3 编码实现

生产者服务 `ProducerServiceImpl` 的实现如下:

```

package com.waylau.spring.jms.queue;

import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

public class ProducerServiceImpl implements ProducerService {

    private JmsTemplate jmsTemplate;

    /**
     * 向指定队列发送消息
     */
    public void sendMessage(Destination destination, final String msg) {
        System.out.println("ProducerService 向队列"
            + destination.toString() + "发送了消息: \t" + msg);
        jmsTemplate.send(destination, new MessageCreator() {
            public Message createMessage(Session session) throws JMSException {
                return session.createTextMessage(msg);
            }
        });
    }

    /**
     * 向默认队列发送消息

```

```

    */
    public void sendMessage(final String msg) {
        String destination = jmsTemplate.getDefaultDestination().toString();
        System.out.println("ProducerService 向队列"
            + destination + "发送了消息: \t" + msg);
        jmsTemplate.send(new MessageCreator() {
            public Message createMessage(Session session) throws JMSException {
                return session.createTextMessage(msg);
            }
        });
    }

    public void sendMessage(Destination destination,
        final String msg, final Destination response) {
        System.out.println("ProducerService 向队列"
            + destination + "发送了消息: \t" + msg);
        jmsTemplate.send(destination, new MessageCreator() {
            public Message createMessage(Session session) throws JMSException {
                TextMessage textMessage = session.createTextMessage(msg);
                textMessage.setJMSReplyTo(response);
                return textMessage;
            }
        });
    }

    public void setJmsTemplate(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }
}

```

消费者服务 `ConsumerServiceImpl` 的实现如下:

```

package com.waylau.spring.jms.queue;

import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.TextMessage;

```

```

import org.springframework.jms.core.JmsTemplate;

public class ConsumerServiceImpl implements ConsumerService {

    private JmsTemplate jmsTemplate;

    /**
     * 接收消息
     */
    public void receive(Destination destination) {
        TextMessage tm = (TextMessage) jmsTemplate.receive(destination);
        try {
            System.out.println("ConsumerService 从队列"
                + destination.toString() + "收到了消息: \t" + tm.getText());
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }

    public void setJmsTemplate(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }

}

```

我们在应用中也定义了多种侦听器。

比如消费者侦听器：

```

public class ConsumerListener {

    public String receiveMessage(String message) {
        System.out.println("ConsumerListener 接收到一个 Text 消息: \t" + message);

        return "I am ConsumerListener response";
    }

}

```

消息队列侦听器：

```

public class QueueMessageListener implements MessageListener {

```

```
public void onMessage(Message message) {
    TextMessage tm = (TextMessage) message;
    try {
        System.out.println("ConsumerMessageListener 收到了文本消息: \t" +
tm.getText());
    } catch (JMSEException e) {
        e.printStackTrace();
    }
}
```

会话感知侦听器:

```
package com.waylau.spring.jms.queue;

import javax.jms.Destination;
import javax.jms.JMSEException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.springframework.jms.listener.SessionAwareMessageListener;

public class ConsumerSessionAwareMessageListener
    implements SessionAwareMessageListener<TextMessage> {

    private Destination destination;

    public void onMessage(TextMessage message, Session session) throws
JMSEException {
        // 接收消息
        System.out.println("SessionAwareMessageListener 收到一条消息: \t" +
message.getText());

        // 发送消息
        MessageProducer producer = session.createProducer(destination);
        TextMessage tm =
```



```
        session.createTextMessage("I am ConsumerSessionAware-  
MessageListener");  
        producer.send(tm);  
    }  
  
    public void setDestination(Destination destination) {  
        this.destination = destination;  
    }  
}
```

为了演示订阅功能，我们也定义了主题提供者 and 主题侦听器。

主题提供者如下：

```
package com.waylau.spring.jms.topic;  
  
import javax.jms.Destination;  
import javax.jms.JMSException;  
import javax.jms.Message;  
import javax.jms.Session;  
  
import org.springframework.jms.core.JmsTemplate;  
import org.springframework.jms.core.MessageCreator;  
  
public class TopicProvider {  
  
    private JmsTemplate topicJmsTemplate;  
  
    /**  
     * 向指定的 Topic 发布消息  
     *  
     * @param topic  
     * @param msg  
     */  
    public void publish(final Destination topic, final String msg) {
```

```

        topicJmsTemplate.send(topic, new MessageCreator() {
            public Message createMessage(Session session) throws JMSEException {
                System.out.println("TopicProvider 发布了主题: \t"
                    + topic.toString() + ", 发布消息内容为:\t" + msg);
                return session.createTextMessage(msg);
            }
        });
    }

    public void setTopicJmsTemplate(JmsTemplate topicJmsTemplate) {
        this.topicJmsTemplate = topicJmsTemplate;
    }
}

```

主题侦听器如下:

```

package com.waylau.spring.jms.topic;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

public class TopicMessageListener implements MessageListener {

    public void onMessage(Message message) {
        TextMessage tm = (TextMessage) message;
        try {
            System.out.println("TopicMessageListener 监听到消息: \t" + tm.getText());
        } catch (JMSEException e) {
            e.printStackTrace();
        }
    }
}

```

主题侦听器 2 如下:

```
package com.waylau.spring.jms.topic;

import javax.jms.JMSException;

import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

public class TopicMessageListener2 implements MessageListener {

    public void onMessage(Message message) {
        TextMessage tm = (TextMessage) message;
        try {
            System.out.println("TopicMessageListener2 监听到消息 \t" + tm.getText());
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }

}
```

3.6.4 运行

为了方便测试，我们编写了如下测试用例：

```
package com.waylau.spring.jms;

import javax.jms.Destination;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import com.waylau.spring.jms.queue.ConsumerService;
```

```
import com.waylau.spring.jms.queue.ProducerService;
import com.waylau.spring.jms.topic.TopicProvider;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/spring.xml")
public class SpringJmsTest {

    /**
     * 队列名 queue1
     */
    @Autowired
    private Destination queueDestination;

    /**
     * 队列名 queue2
     */
    @Autowired
    private Destination queueDestination2;

    /**
     * 队列名 sessionAwareQueue
     */
    @Autowired
    private Destination sessionAwareQueue;

    /**
     * 队列名 adapterQueue
     */
    @Autowired
    private Destination adapterQueue;

    /**
     * 主题 guo_topic
     */
    @Autowired
    @Qualifier("topicDestination")
    private Destination topic;
```



```
/**
 * 主题消息发布者
 */
@Autowired
private TopicProvider topicProvider;

/**
 * 队列消息生产者
 */
@Autowired
@Qualifier("producerService")
private ProducerService producer;

/**
 * 队列消息消费者
 */
@Autowired
@Qualifier("consumerService")
private ConsumerService consumer;

/**
 * 测试生产者向 queue1 发送消息
 */
@Test
public void testProduce() {
    String msg = "Hello world!";
    producer.sendMessage(msg);
}

/**
 * 测试消费者从 queue1 接收消息
 */
@Test
public void testConsume() {
    consumer.receive(queueDestination);
}
```

```
/**
 * 测试消息监听
 * 1.生产者向队列 queue2 发送消息
 * 2.ConsumerMessageListener 监听队列，并消费消息
 */
@Test
public void testSend() {
    producer.sendMessage(queueDestination2, "Hello R2");
}

/**
 * 测试主题监听
 * 1.生产者向主题发布消息
 * 2.ConsumerMessageListener 监听主题，并消费消息
 */
@Test
public void testTopic() {
    topicProvider.publish(topic, "Hello Topic!");
}

/**
 * 测试 SessionAwareMessageListener
 * 1.生产者向队列 sessionAwareQueue 发送消息
 * 2. SessionAwareMessageListener 接收消息，并向 queue1 队列发送回复消息
 * 3. 消费者从 queue1 消费消息
 */
@Test
public void testAware() {
    producer.sendMessage(sessionAwareQueue, "Hello sessionAware");
    consumer.receive(queueDestination);
}

/**
 * 测试 MessageListenerAdapter
 * 1.生产者向队列 adapterQueue 发送消息
```

```

* 2. ConsumerListener 接收消息，并向 queue1 队列发送回复消息
* 3. 消费者从 queue1 消费消息
*
*/
@Test
public void testAdapter() {
    producer.sendMessage(adapterQueue, "Hello adapterQueue", queueDestination);
    consumer.receive(queueDestination);
}
}

```

先启动 ActiveMQ 服务，再执行该测试用例。可以在控制台看到如下输出信息：

```

INFO | Successfully connected to tcp://localhost:61616
INFO | Successfully connected to tcp://localhost:61616
INFO | Successfully connected to tcp://localhost:61616
INFO | Successfully connected to tcp://localhost:61616
INFO | Successfully connected to tcp://localhost:61616
ProducerService 向队列 queue://adapterQueue 发送了消息: Hello adapterQueue
INFO | Successfully connected to tcp://localhost:61616
ConsumerListener 接收一个 Text 消息: Hello adapterQueue
INFO | Successfully connected to tcp://localhost:61616
ConsumerService 从队列 queue://queue1 收到了消息: I am ConsumerListener response
ProducerService 向队列 queue://sessionAwareQueue 发送了消息: Hello sessionAware
INFO | Successfully connected to tcp://localhost:61616
SessionAwareMessageListener 收到一条消息: Hello sessionAware
INFO | Successfully connected to tcp://localhost:61616
ConsumerService 从队列 queue://queue1 收到了消息: I am ConsumerSessionAware-
MessageListener
INFO | Successfully connected to tcp://localhost:61616
TopicProvider 发布了主题: topic://guo_topic, 发布消息内容为 Hello Topic!
TopicMessageListener2 监听到消息 Hello Topic!
TopicMessageListener 监听到消息: Hello Topic!
ProducerService 向队列 queue://queue2 发送了消息: Hello R2
INFO | Successfully connected to tcp://localhost:61616
ConsumerMessageListener 收到了文本消息: Hello R2
ProducerService 向队列 queue://queue1 发送了消息: Hello world!

```

```
INFO | Successfully connected to tcp://localhost:61616  
INFO | Successfully connected to tcp://localhost:61616  
ConsumerService 从队列 queue://queue1 收到了消息: Hello world!
```

上面例子中的代码可以在 <https://github.com/waylau/distributed-systems-technologies-and-cases-analysis> 的 jms-msg 程序中找到。



第 4 章

分布式计算

4.1 分布式计算概述

在过去的 20 年里，互联网产生了大量的数据，比如爬虫文档、Web 请求日志等；也包括各种计算类型的派生数据，比如倒排索引、Web 文档的图结构的各种表示、每台主机页面数量的概要、每天被请求数量最多的集合，等等。这些数据每天需要通过大量的计算产生，显然在单机上是无法做到的，必须使用分布式计算。

分布式计算在概念上非常好理解。举例来说，在工厂里面要生产一批货物，一个工人来干，需要 10 天才能完成。那么，在工人的工作效率相等的情况下，把相同的活派给多个工人来干，显然能够缩短整个工期。分布式计算也是如此，当输入的数据量很大时，这些计算必须分摊到多台机器上才有可能在可以接受的时间内完成。机器越多，所需要的总时间越短。这就是分布式计算带来的优势——通过扩展机器数实现计算能力的水平扩展。

所谓分布式计算，就是将需要大量计算的项目数据分割成小块，由多台计算机分别计算，在上传运算结果后统一合并得出数据结论。

设计分布式计算平台需要面临非常多的挑战，比如分布式平台是如何实现并行计算的？是如何分发数据的？又是如何进行错误处理的？这些问题综合在一起，使得原本很简捷的计算，因为需要使用大量的复杂代码来处理这些问题，而变得让人难以处理。

好在目前市面上已经有了很多分布式计算产品可供选择，本书会对这些产品一一介绍。

4.1.1 使用场景

是否需要使用分布式计算，需要根据项目的业务情况而定。虽然分布式计算可以使整体的计算能力实现水平扩展，但并非所有的计算任务都需要分布式计算平台来解决。比如在 Oracle 数据库中，有一百万条工人的薪资单数据，我们要统计这些工人的薪资的总和。在这种情况下，直接在 PL/SQL 中执行 SUM 函数显然要快于导入分布式计算平台中执行。如果数据量在 TB 级别，那么最好采用分布式计算，毕竟 Oracle 等关系型数据库并不擅长大数据计算。

同时，使用分布式计算需要一定的学习成本，而且一般的企业也不大可能拥有用于分布式计算的大规模的机器数量。这时使用现在的分布式计算云服务可能是最经济的分布式计算方式。阿里云、腾讯云、华为云等都提供了类似的用于分布式计算的云服务。

4.1.2 常用技术

Google 作为世界领先的科技公司，为了应对大数据处理，内部已经实现了数以百计的为专



门目的而写的计算程序。而在开源界，基于 MapReduce 思想的分布式计算产品也很多，比较著名的有 Apache Hadoop、Apache Spark、Apache Mesos 等。本书接下来将会对这些技术进行详细介绍。

4.2 MapReduce

MapReduce 是 Google 出品的著名的计算框架之一，与 GFS、Bigtable 一起被称为 Google 技术的“三宝”。

4.2.1 MapReduce 简介

MapReduce 是一个编程模型，用于大规模数据集（TB 级）的并行运算。

MapReduce 程序模型应用的成功要归功于以下几个方面。首先，由于该模型隐藏了并行、容错、本地优化及负载平衡的细节，所以即便是那些没有并行和分布式系统经验的程序员也易于使用该模型。其次，MapReduce 计算可以很容易地表达大数据的各种问题。比如，MapReduce 用于为 Google 的网页搜索服务生成数据，用于排序、数据挖掘，以及机器学习及其他许多系统。再次，MapReduce 的实现符合“由数千台机器组成的大集群”的尺度，有效地利用了机器资源，所以非常适合解决许多大型计算问题。

4.2.2 MapReduce 的编程模型

MapReduce 是一个用于大规模数据集（TB 级）并行运算的编程模型，其基本原理就是将大的数据分成小块逐个分析，最后将提取出来的数据汇总分析，最终获得我们想要的内容。从名字可以看出，“Map（映射）”和“Reduce（归约）”是 MapReduce 模型的核心，其灵感来源于函数式语言（比如 Lisp）中的内置函数 map 和 reduce：用户通过定义一个 map 函数，处理 key/value（键值对）以生成一个中间 key/value 集合，MapReduce 库将所有拥有相同的 key（key I）的中间状态 key 合并起来传递到 Reduce 函数；Reduce 函数合并所有先前 map 函数处理过后的有相同 key（key I）的中间量。

```
map (k1,v1) -> list(k2,v2)
reduce (k2,list(v2)) -> list(k3, v3)
```

但上面的定义显然还是过于抽象。Shekhar Gulati 在他的博客里面 *How I explained MapReduce to my Wife?*（博客地址见 <http://www.pixelstech.net/article/1314519773-How-I-explained-MapReduce-to-my-Wife>）举了一个制作辣椒酱的例子，来形象地描述 MapReduce 的原



理，如下文所述。

1. MapReduce 制作辣椒酱的过程

制作辣椒酱的过程是这样的，先取一个洋葱，把它切碎，然后拌入盐和水，最后放进研磨机里研磨。这样就能得到洋葱辣椒酱了。

现在，假设你想用薄荷、洋葱、番茄、辣椒、大蒜弄一瓶混合辣椒酱。你会怎么做呢？你会取薄荷叶一撮、洋葱一个、番茄一个、辣椒一根、大蒜一头，切碎后加入适量的盐和水，再放入研磨机里研磨，这样就能得到一瓶混合辣椒酱了。

现在把 MapReduce 的概念应用到食谱上，Map 和 Reduce 其实是两种操作。

- **Map:** 把洋葱、番茄、辣椒和大蒜切碎，是各自作用在这些物体上的一个 Map 操作。所以你给 Map 一个洋葱，Map 就会把洋葱切碎。同样，你把辣椒、大蒜和番茄一一拿给 Map，也会得到各种碎块。所以，当你在切像洋葱这样的蔬菜时，你执行的就是一个 Map 操作。Map 操作适用于每一种蔬菜，它会相应地生产出一种或多种碎块，在我们的例子中生产的是蔬菜块。在 Map 操作中可能会出现某个洋葱烂掉的情况，则只需把烂洋葱丢掉即可。所以，如果出现烂洋葱，则 Map 操作就会过滤掉这个烂洋葱而不生产任何坏洋葱块。
- **Reduce:** 在这一阶段，你将各种蔬菜都放入研磨机里进行研磨，就可以得到一瓶辣椒酱了。这意味着要制成一瓶辣椒酱，你需要研磨所有的原料。因此，研磨机通常将 Map 操作的蔬菜聚集在了一起。

当然，上面的内容只是 MapReduce 的一部分，MapReduce 的强大在于分布式计算。假设你每天需要生产 10000 瓶辣椒酱，应该怎么办呢？这个时候你就需要用更多的人和研磨机来完成这项工作。你需要几个人一起切蔬菜。每个人都要处理满满一袋的蔬菜，而每个人都相当于在执行一个简单的 Map 操作。每个人都将不断地从袋子里拿出蔬菜，并且每次只对一种蔬菜进行处理，也就是将它们切碎，直到袋子空了为止。这样，当所有的工人都切完以后，工作台（每个人工作的地方）上就有了洋葱块、番茄块和蒜蓉，等等。

MapReduce 将所有输出的蔬菜都搅拌在了一起，这些蔬菜都是在以 key 为基础的 Map 操作下产生的。搅拌将自动完成，你可以假设 key 是一种原料的名字，就像洋葱一样。所以全部的洋葱 key 都会搅拌在一起，并转移到研磨洋葱的研磨器里。这样，你就能得到洋葱辣椒酱了。同样，所有的番茄也会被转移到标记着番茄的研磨器里，并制造出番茄辣椒酱。

2. 统计词频的例子

考虑这样一个问题：以一大堆文档为单位查找某个词语出现的词频。用户可能像下面这样书写伪代码：



```
map(String key, String value):  
    // key: 文档的名称  
    // value: 文档的内容  
    for each word w in value:  
        EmitIntermediate(w, "1");  
  
reduce(String key, Iterator values):  
    // key: 单词  
    // values: 词频列表  
    int result = 0;  
    for each v in values:  
        result += ParseInt(v);  
    Emit(AsString(result));
```

在统计词频的例子中，map 函数接收的键是文件名，值是文件的内容，map 函数逐个遍历单词，每遇到一个单词 *w* 就产生一个中间 key/value 对 $\langle w, "1" \rangle$ ，这表示又找到一个单词 *w*。MapReduce 将 key 相同（都是单词 *w*）的 key/value 对传给 reduce 函数，这样 reduce 函数接收的 key 就是单词 *w*，值是一串“1”（最基本的实现是这样的，但可以优化），个数是 key 为 *w* 的 key/value 对的个数，然后将这些“1”累加就能得到单词 *w* 的出现次数。最后，这些单词的出现次数会被写到用户定义的位置，存储在底层的分布式存储系统（GFS 或 HDFS）中。

下面是一个完整的用 C++ 编写的统计词频的代码例子：

```
#include "mapreduce/mapreduce.h"  
  
// 用户的 map 函数  
class WordCounter : public Mapper {  
public:  
    virtual void Map(const MapInput& input) {  
        const string& text = input.value();  
        const int n = text.size();  
        for (int i = 0; i < n; ) {  
            // 跳过前面的空格  
            while ((i < n) && isspace(text[i]))  
                i++;  
  
            // 查找单词  
            int start = i;  
            while ((i < n) && !isspace(text[i]))
```



```
        i++;

        if (start < i)
            Emit(text.substr(start, i-start), "1");
    }
}

};

REGISTER_MAPPER(WordCounter);

// 用户的 reduce 函数
class Adder : public Reducer {
    virtual void Reduce(ReduceInput* input) {
        // 遍历相同的 key 和相同的数据项
        int64 value = 0;
        while (!input->done()) {
            value += StringToInt(input->value());
            input->NextValue();
        }

        // Emit sum for input->key()
        Emit(IntToString(value));
    }
};

REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
    ParseCommandLineFlags(argc, argv);

    MapReduceSpecification spec;

    // 存储输入的文件列 "spec"
    for (int i = 1; i < argc; i++) {
        MapReduceInput* input = spec.add_input();
        input->set_format("text");
        input->set_filepattern(argv[i]);
        input->set_mapper_class("WordCounter");
    }
}
```



```
// 指定输出的文件:
//    /gfs/test/freq-00000-of-00100
//    /gfs/test/freq-00001-of-00100
//    ...
MapReduceOutput* out = spec.output();
out->set_filebase("/gfs/test/freq");
out->set_num_tasks(100);
out->set_format("text");
out->set_reducer_class("Adder");

// 可选: 在 map 任务中执行部分求和
// 节省网络带宽
out->set_combiner_class("Adder");

// 调整参数: 每个任务最多使用 2000 个机器和 100 MB 内存
spec.set_machines(2000);
spec.set_map_megabytes(100);
spec.set_reduce_megabytes(100);

// 现在运行它
MapReduceResult result;
if (!MapReduce(spec, &result)) abort();

// 完成: 'result' 结构里面包含了相关信息
// 包括计数器、执行时间、所使用的机器数, 等等

return 0;
}
```

3. 类型

前面例子的代码展示的是输入/输出字符串, 由用户定义的 map 和 reduce 函数还支持以下类型:

```
map (k1,v1) -> list(k2,v2)
reduce (k2,list(v2)) -> list(v2)
```

也就是说, 输入的 key/value 是从相对于输出 key/value 的不同域中提取的, 中间 key/value 和输出 key/value 都来自同一个域。



4. 更多的例子

下面列举了一些简单的程序，这些程序很清晰地表达了 MapReduce 的计算应用场景。

- **分布式 grep:** map 函数在匹配一个给定模式时会产生一行。同时 reduce 函数是一种身份拣选机制，将中间 key/value 对复制操作后转化为输出 key/value 对。
- **URL 访问词频统计:** map 函数处理网页日志请求及输出<URL, 1>。reduce 函数将所有属于同一个 URL 的 value 进行叠加并输出<URL, total count>。
- **倒转网络链接图:** map 函数为每一个链接输出<target, source> key/value 对，一个 URL 叫作 target，包含这个 URL 的页面叫作 source。reduce 函数将所有关联 target URL 的源 URL 列表进行拼接并产生<target, list(source)> key/value 对。
- **每台主机的词汇向量:** 词汇向量总结了出现在一个文档中的最重要单词，或者总结了出现在一个文档集合中的最重要单词，这个文档集合被表示成<word, frequency>列表。map 函数对每个输入的文档产生<hostname, term vector> key/value 对（这里的主机名 hostname 是从输入文档的 URL 中提取的）。这些 key/value 对被传递给了 reduce 函数。reduce 函数将这些词汇变量汇总起来，丢弃非高频词汇并最终产生 key/value 对<hostname, term vector>。
- **倒排索引:** map 函数解析每个文档，同时产出<word, document, ID>的一个序列。reduce 函数接收给定单词的所有 key/value 对，对相应的文档 ID 进行排序，同时产生<word, list(document ID)> key/value 对。所有输出的 key/value 对的集合形成了简单的倒排索引。通过追踪单词的位置来使计算结果累加。
- **分布式排序:** map 函数将每个 key 从记录中提取出来，产生<key, record> key/value 对。reduce 函数产生所有未经改变的 key/value 对。

4.2.3 MapReduce 接口实现

MapReduce 接口可以有多种不同实现，需要根据不同的环境选择正确的接口实现。比如，其中一种 MapReduce 接口实现可能适用于小型共享内存机器，另一种 MapReduce 接口实现则适用于多处理器 NUMA（Non-Uniform Memory Access Architecture），还有一种 MapReduce 接口实现的适用情况是大规模网络机器的集合。

下面将介绍在 Google 内广泛使用的一种 MapReduce 接口实现——由交换以太网连接起来的商用 PC 组成的大规模集群，该系统的环境如下。

- 典型的单台机器硬件环境是 x86 双核处理器，2~4GB 内存，操作系统为 Linux。





- 典型的商用网络硬件在机器级别为 100Mbps 或 1Gbps，但均值通常明显低于带宽的一半。
- 一个集群由数百或数千台机器构成，这种架构必然带来 Machine Failures（机器失效）的问题。
- 存储设备由不怎么昂贵的 IDE 磁盘组成（同台式 PC）。一个 Google 内部开发的分布式系统用于管理存储于这些机器磁盘上的数据。
- 用户将 job（作业）提交至 scheduling system（调度系统）。每个 job 由一系列任务集合组成，都由调度器通过映射存储于一个集群的可用机器集合中。

1. MapReduce 概述

通过自动分割将输入数据分成一个有 M 个 split 的集，map 调用被分布到多台机器上。输入的 split 能够在不同的机器上被并行处理。通过分割函数分割中间 key，来形成 R 个片（例如， $\text{hash}(\text{key}) \bmod R$ ），reduce 调用被分布到多台机器上。分割数量（ R ）和分割函数由用户指定。

图 4-1 显示了 MapReduce 操作的流程。

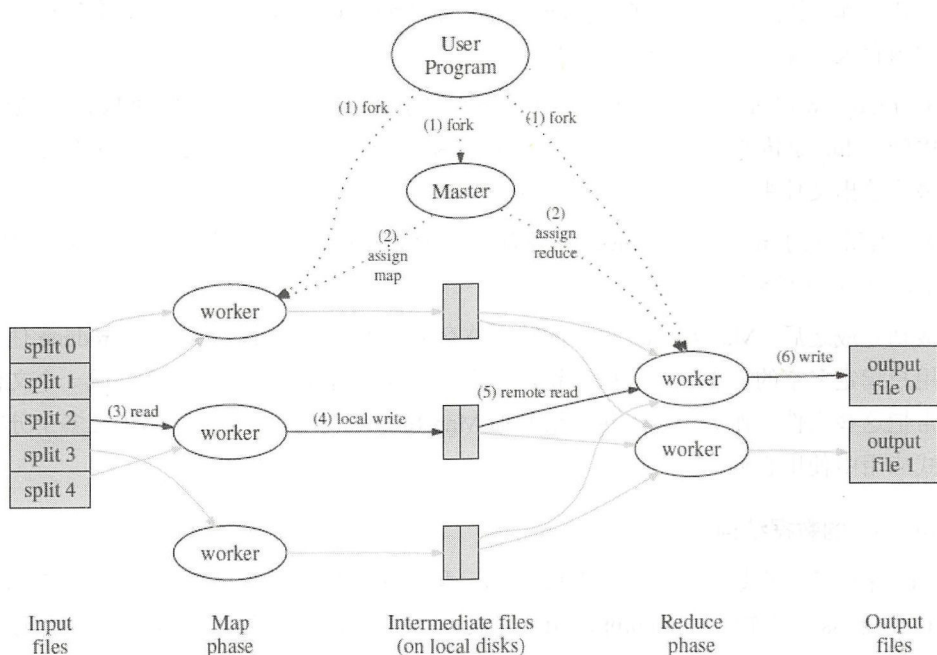


图 4-1 MapReduce 的操作流程

当用户的程序调用 MapReduce 的函数时，将发生下面的一系列动作（下面的序号和图 4-1 中的数字标签相对应）：





(1) 用户程序里的 MapReduce 库首先将输入文件分割成 M 个片，每个片的大小一般为 16~64 MB（用户可以通过可选的参数来控制）。然后在机群中大量复制程序。

(2) 在这些程序副本中，一个是 master，其他都是由 master 分配任务的 worker。有 M 个 map 任务和 R 个 reduce 任务将被分配。master 分配一个 map 任务或 reduce 任务给一个空闲的 worker。

(3) 一个被分配了 map 任务的 worker 读取相关输入 split 的内容。它从输入数据中分析出 key/value 对，然后把 key/value 对传递给用户自定义的 map 函数。由 map 函数产生的中间 key/value 对被缓存在内存中。

(4) 缓存在内存中的 key/value 对被周期性地写入本地磁盘，通过分割函数把它们写入 R 个区域。本地磁盘上的缓存对的位置被传送给 master，master 负责把这些位置传送给负责 reduce 的 worker。

(5) 一个 reduce worker 在得到 master 的位置通知时，使用远程过程调用来从 map worker 的磁盘上读取缓存数据。reduce worker 在读取所有的中间数据后，通过排序使具有相同 key 的内容聚合在一起。因为许多不同的 key 映射到相同的 reduce 任务，所以排序是必需的。如果中间数据比内存大，那么还需要一个外部排序。

(6) reduce worker 迭代排过序的中间数据，对于遇到的每个唯一的中间 key，会把 key 和相关的中间 value 集传递给用户自定义的 reduce 函数。reduce 函数的输出被添加到这个 reduce 分割的最终输出文件中。

(7) 当所有的 map 和 reduce 任务都完成时，master 唤醒用户程序，用户程序里的 MapReduce 调用返回到用户代码。

在成功完成之后，MapReduce 执行的输出被存放在 R 个输出文件中（每个 reduce 任务产生一个由用户指定名字的文件）。一般来说，用户不需要将这 R 个输出文件合并成一个文件——用户经常把这些文件当作一个输入传递给其他 MapReduce 调用，或者在可以处理多个分割文件的分布式应用中使用它们。

2. master 的数据结构

master 保持了一些数据结构。它为每个 map 和 reduce 任务存储状态，其状态包括 idle（空闲）、in-progress（工作中）和 completed（完成）；同时存储了 worker 机器（非空闲任务的机器）的标识。

master 就像一个管道，通过它，中间文件区域的位置从 map 任务传递到 reduce 任务。因此，对于每个完成的 map 任务，master 存储由 map 任务产生的 R 个中间文件区域的大小和位置。当 map 任务完成的时候，位置和大小更新信息被接收。这些信息被逐步地传递给那些正在工作





的 reduce 任务。

3. 容错

因为 MapReduce 库被设计用来使用成百上千的机器处理大规模数据，所以这个库必须能很好地处理机器故障。

worker 失效

master 周期性地 ping 每个 worker。如果 master 在一个确定的时间段内没有收到 worker 的响应信息，那么它将把这个 worker 标记成失效。因为每一个由这个失效的 worker 完成的 map 任务都将被重新设置成它初始的空闲状态，所以它可以被安排给其他 worker。同样，每一个在失败的 worker 上正在运行的 map 或 reduce 任务也被重新设置成 idle 状态，并且将被重新调度。

在一个失效机器上已经完成的 map 任务将被再次执行，因为它的输出存储在它的磁盘上，所以不可访问。已经完成的 reduce 任务将不会被再次执行，因为它的输出存储在全局文件系统中。

当一个 map 任务首先被 workerA 执行之后，由于 A 失效了，所以又被 B 执行了，重新执行这个情况被通知给所有执行 reduce 任务的 worker。任何还没有从 A 中读数据的 reduce 任务都将从 worker B 中读取数据。

MapReduce 可以处理大规模 worker 失效的情况。例如，在一个 MapReduce 操作期间，在运行的机群上进行网络维护引起 80 台机器在几分钟内不可访问，MapReduce master 只是简单地再次执行已经被不可访问的 worker 完成的工作，最终完成这个 MapReduce 操作。

master 失效

让 master 监控线程同期性地设置上文所述的 master 数据结构检查点是很容易的，如果这个 master 任务失效了，则可以从上次最后一个检查点开始启动另一个 master 进程。因为只有一个 master，所以它的失效是比较麻烦的，因此现在的实现是：如果 master 失效，则中止 MapReduce 计算。客户可以检查这个状态，并且可以根据需要重新执行 MapReduce 操作。

在失效面前的处理机制

当用户提供的 map 和 reduce 操作对它的输出值是确定的函数时，我们的分布式实现将产生和全部程序无错状态下顺序执行一样的输出结果。

我们通过 map 和 reduce 任务输出所进行的原子提交来实现上面的特性。每个 in-progress 状态的任务把它的输出写到私有临时文件中。一个 reduce 任务产生一个这样的文件，而一个 map 任务产生 R 个这样的文件（一个 reduce 任务对应一个文件）。当一个 map 任务完成时，worker 会发送一个消息给 master，在这个消息中包含 R 个临时文件的名字。如果 master 从一个已经完成的 map 任务再次收到一个完成的消息，则它将忽略该消息；否则，它会在 master 的数据结构



里记录这 R 个文件的名字。

当一个 reduce 任务完成时, 这个 reduce worker 原子地把临时文件重命名成最终的输出文件。如果相同的 reduce 任务在多个机器上执行, 则多个重命名调用将被执行, 并产生相同的输出文件。我们依赖由底层文件系统提供的原子重命名操作, 来保证最终的文件系统状态仅仅包含一个 reduce 任务产生的数据。

我们的 map 和 reduce 操作大部分都是确定的, 并且我们的处理机制等价于按顺序执行, 这使得程序员可以很容易地理解程序的行为。当 map 或 (和) reduce 操作并不确定时, 我们提供了比较弱但合理的处理机制。在一个非确定操作的前面, 一个 reduce 任务 R_1 的输出等价于一个非确定顺序程序执行产生的输出。然而, 一个不同的 reduce 任务 R_2 的输出也许符合一个不同的非确定顺序程序执行产生的输出。

考虑 map 任务 M 和 reduce 任务 R_1 、 R_2 的情况。我们设定 $e(R_i)$ 为已经提交的 R_i 的执行 (有且仅有一个这样的执行)。出现这个比较弱的语义, 是因为 $e(R_1)$ 也许已经读取了由 M 的执行产生的输出, 而 $e(R_2)$ 也许已经读取了由 M 的不同执行产生的输出。

4. 存储位置

在 Google 的计算机环境里, 网络带宽是一个相当缺乏的资源, 所以会把输入数据 (由 GFS 管理) 存储在机器的本地磁盘上以节省网络带宽。GFS 把每个文件分成 64 MB 的一些块, 然后把每个块的几个副本存储在不同的机器上 (一般是 3 个副本)。MapReduce 的 master 考虑输入文件的位置信息, 并且努力在一个包含相关输入数据的机器上安排一个 map 任务。如果这样做失败了, 则它尝试在那个任务的输入数据的附近安排一个 map 任务, 例如, 分配到一个和包含输入数据块在一个网络交换机里的 worker 机器上执行。当在一个机群中的一部分机器上运行巨大的 MapReduce 操作时, 大部分输入数据都在本地被读取, 从而不消耗网络带宽。

5. 任务粒度

像上面描述的那样, 我们把 map 阶段细分成 M 个片, 把 reduce 阶段分成 R 个片。 M 和 R 应当比 worker 机器的数量大许多。每个 worker 执行许多不同的工作来提高动态负载均衡, 也可以加速从一个 worker 失效中恢复, 这个机器上的许多已经完成的 map 任务可以被分配到所有其他的 worker 机器上。

在我们的实现里, M 和 R 的范围是有大小限制的, 因为 master 必须做 $O(M+R)$ 次调度, 并且保存 $O(M \times R)$ 个状态在内存中 (这个因素使用的内存是很少的, 在 $O(M \times R)$ 个状态片里, 大约每个 map/reduce 任务对使用一个字节的数据)。

此外, R 经常被用户限制, 因为每个 reduce 任务最终都是一个独立的输出文件。实际上, 我们倾向于选择 M , 以便每一个单独的任务大概都是 16~64MB 的输入数据 (以便上面描述的



位置优化是最有效的)，我们把 R 设置成我们希望使用的 worker 机器数量的倍数。Google 经常执行 MapReduce 计算的情况是， M 为 200000， R 为 5000，使用 2000 台 worker 机器。

6. 备用任务

“straggler（落后者）”是延长 MapReduce 操作时间的主要原因之一：一个机器需要花费一个异乎寻常的长时间来完成最后的一些 map 或 reduce 任务中的一个。产生 straggler 的原因有多。例如，一个有坏磁盘的机器经常发生可以纠正的错误，这样就使读性能从 30 MB/s 降低到 3 MB/s。机群调度系统也许已经安排其他的任务在这个机器上，由于计算要使用 CPU、内存、本地磁盘、网络带宽的原因，导致它执行 MapReduce 代码很慢。还有其他方面的问题，比如，一个在机器初始化时的 bug 引起处理器缓存的失效，对机器上的计算性能可能有上百倍的影响。

有一个一般的机制来减轻这个 straggler 导致的影响。当一个 MapReduce 操作将要完成时，master 调度备用进程来执行那些剩下的 in-progress 状态的任务。无论原来的还是备用的进程执行完成了，工作都被标记成完成。这个机制通常只会多占用几个百分点的机器资源，却可以显著地减少完成大规模 MapReduce 操作的时间。在 Google 进行的排序测试中，对 10^{10} 个记录进行排序，每个记录 100 个字节（大概 1 TB 的数据）。测试结果显示，在关闭备用任务的情况下，要比有备用任务的情况下多花 44% 的时间。图 4-2（a）是正常运行的情况，图 4-2（b）是关闭备用任务的执行情况。

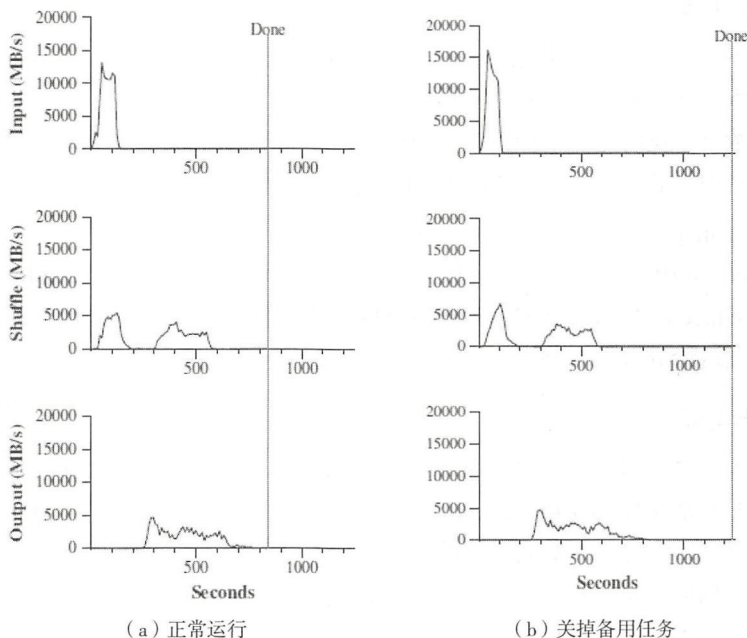


图 4-2 MapReduce 启用备用任务前后的对比



4.2.4 MapReduce 的使用技巧

尽管简单的 Map 和 Reduce 函数的功能对于大多数需求是足够了，但是我们开发了一些有用的扩充，这些内容将在本节描述。

1. 分割函数

MapReduce 用户指定 reduce 任务和 reduce 任务需要的输出文件的数量。一个默认的分割函数是使用 hash 方法（例如， $\text{hash}(\text{key}) \bmod R$ ），从而达到非常平衡的分割。但有些时候，使用其他的 key 分割函数来分割数据也是非常有用的。例如，输出的 key 是 URL，并且我们希望每个主机的所有条目保持在同一个输出文件中。为了支持类似的情况，MapReduce 库的用户可以提供专门的分割函数。例如，使用 $\text{hash}(\text{Hostname}(\text{urlkey})) \bmod R$ 作为分割函数，使所有来自同一个主机的 URL 保存在同一个输出文件中。

2. 顺序保证

MapReduce 保证在一个给定的分割里面，中间 key/value 对以 key 递增的顺序处理。这个顺序可以保证每个分割都能产生一个有序的输出文件，当输出文件的格式需要支持有效率的随机访问 key 的时候，或者对输出数据集再进行排序的时候，就显得很容易。

3. Combiner 函数

在某些情况下，允许中间结果 key 重复会占据相当的比重，每个 map 任务将产生成百上千个这样的记录。所有计数都将通过网络被传输到一个单独的 reduce 任务，然后由 reduce 函数加在一起产生一个数字。我们允许用户指定一个可选的 Combiner 函数，先在本地进行合并，然后通过网络发送，从而显著地提高一些 MapReduce 操作的速度。

在每一个执行 map 任务的机器上，Combiner 函数都会被执行。一般来说，相同的代码被用在 Combiner 和 reduce 函数中。Combiner 和 reduce 函数之间的唯一区别是 MapReduce 库怎样控制函数的输出。reduce 函数的输出被保存在最终的输出文件里。Combiner 函数的输出被写到中间文件里，然后发送给 reduce 任务。

4. 输入/输出类型

MapReduce 库支持以几种不同的格式读取输入数据。例如，文本模式输入把每一行看作一个 key/value 对。key 是文件的偏移量，value 是那一行的内容。其他普通的支持格式以 key 的顺序存储 key/value 对序列。每一个输入类型的实现知道怎样把输入分割成对每个单独的 map 任务来说是有意义的（例如，文本模式的范围分割确保仅在每行的边界进行）。虽然许多用户仅使用很少的预定义输入类型，但是用户可以通过提供一个简单的 reader 接口来支持一个新的输入类型。



一个 reader 不必从文件里读数据。例如，我们可以很容易地定义它从数据库里读取记录，或者从内存的数据结构中读取记录。

5. 副作用

在某些情况下，在 Map 和（或）Reduce 操作过程中增加辅助的输出文件会比较省事。我们依靠程序 writer 把这种“副作用”变成原子的和幂等的。通常应用程序会把输出结果写到一个临时文件中，在输出全部数据之后，再使用系统级的原子操作 rename 重新命名这个临时文件。

如果一个任务产生了多个输出文件，则没有提供类似两阶段提交的原子操作不支持一个任务产生多个输出文件的情况。因此，对于会产生多个输出文件，并且对于跨文件有一致性要求的任务，都必须是确定性的任务。但在实际应用过程中，这个限制还没有给我们带来麻烦。

6. 跳过错误记录

有时因为在用户的代码里有 bug，导致在某一个记录上 map 或 reduce 函数突然“crash”，这样的 bug 使得 MapReduce 操作不能完成。虽然一般可以修复这个 bug，但有时这是不现实的，因为这个 bug 也许是在第三方库里。有时也可以忽略一些记录，例如，在一个大的数据集上进行统计分析。我们提供一种可选的执行模式，在这种模式下，MapReduce 库检测哪些记录引起的 crash，然后跳过那些记录来继续执行程序。

每个 worker 程序安装一个信号处理器来获取内存段异常和总线错误。在调用一个用户自定义的 Map 或 Reduce 操作之前，MapReduce 库把记录的序列号存储在一个全局变量里。如果用户代码产生一个信号，那个信号处理器会发送一个包含序列号的“last gasp”的 UDP 包给 MapReduce 的 master。当 master 不止一次看到同一个记录时，它就会指出，当相关的 map 或 reduce 任务再次执行时，这个记录应当被跳过。

7. 本地执行

调试在 map 或 reduce 函数中的问题有时是很困难的，因为实际的计算发生在一个分布式系统中，经常有一个 master 动态地分配工作给几千台机器。为了简化调试和测试，一个可替换的实现出现，这个实现在本地执行所有的 MapReduce 操作，用户可以控制执行，这样计算可以限制到特定的 map 任务上。用户以一个标志调用他们的程序，然后可以容易地使用他们认为好用的任何调试和测试工具（例如 gdb）。

8. 状态信息

master 运行了一个内置的 HTTP 服务器，并且可以输出一组状态页来供用户使用。状态页显示计算进度，像多少个任务已经完成、多少个还在运行、输入的字节数、中间数据字节数、输出字节数、处理百分比，等等。这个页面也包含标准错误的链接和由每个任务产生的标准输



出的链接。用户可以根据这些数据预测计算需要花费的时间和是否需要更多的资源。当计算比预期要慢得多时，这些页面也可以被用来判断是不是这样的情况。

此外，最上面的状态页显示已经有多少个 worker 失效了，它们失效的时间，以及现在哪个 map 和 reduce 任务正在运行。当试图诊断用户代码里的 bug 时，这个信息非常有用。

9. 计数器

MapReduce 库提供了一个计数器工具，用来计算各种事件发生的次数。例如，用户代码想要计算所有处理的词的个数或被索引的德文文档的数量。

为了使用这个工具，用户代码创建一个命名的计数器对象，然后在 map 或 reduce 函数里适当地增加计数器。例如：

```
Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents):
    for each word w in contents:
        if (IsCapitalized(w)):
            uppercase->Increment();
        EmitIntermediate(w, "1");
```

来自不同 worker 机器上的计数器值被周期性地传送给 master（在 ping 响应里）。master 把来自成功的 map 和 reduce 任务的计数器值加起来，在 MapReduce 操作完成时，把它返回给用户代码。当前计数器的值也被显示在 master 状态页里，以便人们查看实际的计算进度。当计算计数器值时会消除重复执行的影响，避免数据的累加（比如，备用任务的使用等可能会导致重复执行）。

有些计数器值由 MapReduce 库自动维护，比如，被处理的输入 key/value 对的数量和被产生的输出 key/value 对的数量。

计数器工具对于检查 MapReduce 操作的完整性十分有用。例如，在一些 MapReduce 操作中，用户代码也许想要确保输出对的数量完全等于输入对的数量，或者处理过的德文文档的数量在全部被处理的文档数量中属于合理的范围。

4.3 Apache Hadoop

Apache Hadoop 是一个由 Apache 基金会开发的分布式系统基础架构，它可以让用户在不了解分布式底层细节的情况下，开发出可靠、可扩展的分布式计算应用。



Apache Hadoop 框架允许用户使用简单的编程模型来实现计算机集群的大型数据集的分布式处理。它的目的是支持从单一服务器到上千台机器的扩展，充分利用了每台机器所提供的本地计算和存储，而不是依靠硬件来提供高可用性，其本身被设计成在应用层检测和处理故障的库。对于计算机集群来说，其中每台机器的顶层都被设计成可以容错的，以便提供一个可用的服务。

Apache Hadoop 框架最核心的设计就是 HDFS 和 MapReduce。HDFS 为海量的数据提供了存储，而 MapReduce 则为海量的数据提供了计算。

4.3.1 Apache Hadoop 的核心组件

Apache Hadoop 包含以下模块。

- Hadoop Common——常见的实用工具，用来支持其他 Hadoop 模块。
- Hadoop Distributed File System (HDFS)——分布式文件系统，它提供对应用程序数据的高吞吐量访问。
- Hadoop YARN——一个作业调度和集群资源管理框架。
- Hadoop MapReduce——基于 YARN 的大型数据集的并行处理系统。

其他与 Apache Hadoop 相关的项目如下。

- Ambari——一个基于 Web 的工具，用于配置、管理和监控 Apache Hadoop 集群，支持 Hadoop HDFS、Hadoop MapReduce、Hive、HCatalog、HBase、ZooKeeper、Oozie、Pig 和 Sqoop。Ambari 还提供了仪表盘（如热图）用于查看集群的“健康状况”，并能够以用户友好的方式来查看 MapReduce、Pig 和 Hive 应用，方便诊断其性能。
- Avro——数据序列化系统。
- Cassandra——可扩展的、无单点故障的多主数据库。
- Chukwa——数据采集系统，用于管理大型分布式系统。
- HBase——一个可扩展的分布式数据库，支持结构化数据的大表存储（有关 HBase 的内容，会在后面的章节中讲述）。
- Hive——数据仓库基础设施，提供数据汇总及特定的查询。
- Mahout——一个可扩展的机器学习和数据挖掘库。
- Pig——一个高层次的数据流并行计算语言和执行框架。
- Spark——Hadoop 数据的快速和通用计算引擎。Spark 提供了简单且强大的编程模型，



用以支持广泛的应用，其中包括 ETL、机器学习、流处理和图形计算（有关 Spark 的内容，会在后面的章节中讲述）。

- **TEZ**——通用的数据流编程框架，建立在 Hadoop YARN 之上。它提供了一个强大而灵活的引擎来执行任意的 DAG 任务，以实现批量和交互式数据的处理。TEZ 目前被 Hive、Pig 和 Hadoop 生态系统中的其他框架所采用。也可以通过其他商业软件（例如，ETL 工具），以取代 Hadoop MapReduce 作为底层执行引擎。
- **ZooKeeper**——一个高性能的分布式应用程序协调服务（有关 ZooKeeper 的内容会在后面章节讲述）。

4.3.2 例子：词频统计 WordCount 程序

Hadoop 官网提供的词频统计 WordCount 程序示例，我们可以拿来运行。例子见 <http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html> 页面的“Example: WordCount v2.0”小节。在运行该程序之前，请确保 HDFS 已经启动。

待输入的样本文件如下：

```
$ bin/hadoop fs -ls /user/joe/wordcount/input/  
/user/joe/wordcount/input/file01  
/user/joe/wordcount/input/file02  
  
$ bin/hadoop fs -cat /user/joe/wordcount/input/file01  
Hello World, Bye World!  
  
$ bin/hadoop fs -cat /user/joe/wordcount/input/file02  
Hello Hadoop, Goodbye to hadoop.
```

运行程序：

```
$ bin/hadoop jar wc.jar WordCount2 /user/joe/wordcount/input /user/joe/  
wordcount/output
```

输出如下：

```
$ bin/hadoop fs -cat /user/joe/wordcount/output/part-r-00000  
Bye 1  
Goodbye 1  
Hadoop, 1  
Hello 2  
World! 1
```

```
World, 1
hadoop. 1
to 1
```

通过 DistributedCache 来设置单词过滤的策略：

```
$ bin/hadoop fs -cat /user/joe/wordcount/patterns.txt
\
\,
\!
to
```

再次运行，这次增加了更多的选项：

```
$ bin/hadoop jar wc.jar WordCount2 -Dwordcount.case.sensitive=true /user/
joe/wordcount/input /user/joe/wordcount/output -skip /user/joe/wordcount/
patterns.txt
```

输出如下：

```
$ bin/hadoop fs -cat /user/joe/wordcount/output/part-r-00000
Bye 1
Goodbye 1
Hadoop 1
Hello 2
World 2
hadoop 1
```

再次运行，这次去掉了大小写敏感：

```
$ bin/hadoop jar wc.jar WordCount2 -Dwordcount.case.sensitive=false /user/
joe/wordcount/input /user/joe/wordcount/output -skip /user/joe/wordcount/
patterns.txt
```

输出如下：

```
$ bin/hadoop fs -cat /user/joe/wordcount/output/part-r-00000
bye 1
goodbye 1
hadoop 2
hello 2
horld 2
```

4.4 Spark

Spark 是一个快速和通用的集群计算系统，提供了 Java、Scala、Python 和 R 等语言的高级 API，并支持通用执行图计算。它还支持一组丰富的更高级别的工具，包括执行 SQL 和结构化数据的处理的 Spark SQL、机器学习的 MLlib、进行图形处理的 GraphX，以及 Spark Streaming。

4.4.1 Spark 简介

1. 快速

Spark 具有支持循环数据流和内存计算的先进的 DAG 执行引擎，所以在内存计算上比 Hadoop 快 100 倍，在硬盘计算上快 10 倍，如图 4-3 所示。

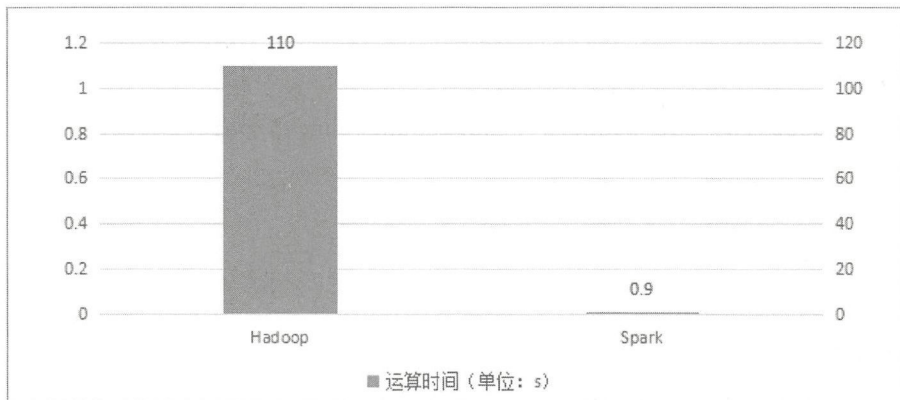


图 4-3 MapReduce 的计算对比

2. 易于使用

Spark 提供了 Java、Scala、Python 和 R 等语言的高级 API，可以用于快速开发相关语言的应用。

Spark 提供了 80 多个高级操作，可以轻松构建并行应用程序。

下面是一段词频统计的程序示例，使用的是 Spark Python 的 API：

```
text_file = spark.textFile("hdfs://...")

text_file.flatMap(lambda line: line.split())
           .map(lambda word: (word, 1))
           .reduceByKey(lambda a, b: a+b)
```


3. 全面

Spark 提供了 Spark SQL、MLlib、GraphX，以及 Spark Streaming 等库。你可以在同一应用程序无缝地合并这些库，如图 4-4 所示。

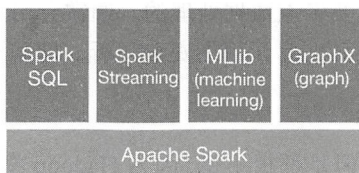


图 4-4 Spark 的组织

4. 到处运行

可以使用它的 standalone cluster mode 运行 EC2、Hadoop YARN 或 Apache Mesos。可以访问 HDFS、Cassandra、HBase、Hive、Tachyon，以及任意的 Hadoop 数据源。

4.4.2 Spark 与 Hadoop 的关系

下面从以下几方面来比较 Spark 与 Hadoop。

1. 解决问题的层面不同

Spark 与 Hadoop 都是大数据框架，但是各自存在的目的不尽相同。Hadoop 实质上更多的是一个分布式数据基础设施：它将巨大的数据集分派到一个由普通计算机组成的集群中的多个节点进行存储，这意味着我们不需要购买和维护昂贵的服务器硬件。

同时，Hadoop 会索引和跟踪这些数据，让大数据处理和分析效率达到前所未有的高度。Spark 则是一个专门用来对那些分布式存储的大数据进行处理的工具，它并不会进行分布式数据的存储。

2. 数据处理方式不同

因为处理数据的方式不一样，Spark 会比 Hadoop 快很多。Hadoop 是分步对数据进行处理，先从集群中读取数据，进行一次处理，将结果写入集群，从集群中读取更新后的数据，再进行下一次处理，将结果写入集群中。

而反观 Spark，它会在内存中接近“实时”地完成所有的数据分析，在集群中读取数据，完成所有必需的分析处理，将结果写回集群，整个计算过程就完成了。所以 Spark 的批处理速度比 Hadoop 快了近 10 倍，内存中的数据分析速度则快了近 100 倍。

如果需要处理的数据和结果需求在大部分情况下是静态的，而且你有耐心等待批处理完成，

则 Hadoop MapReduce 的处理方式也是可以接受的。如果你需要对流数据进行分析，比如那些从工厂的传感器收集回来的数据，或者说你的应用是需要多重数据处理的，那么你也许更应该使用 Spark 进行处理。

大部分机器学习算法都是需要多重数据处理的。通常会用到 Spark 的应用场景有以下几个方面：实时的市场活动、在线产品推荐、网络安全分析、机器日记监控等。

3. 容灾处理

两者的灾难恢复方式迥异，但是都很不错。因为 Hadoop 将每次处理后的数据都写入磁盘，所以其天生就能很有弹性地对系统错误进行处理。Spark 的数据对象存储在分布于数据集群中的弹性分布式数据集（RDD，Resilient Distributed Dataset）中。这些数据对象既可以放在内存中，又可以放在磁盘中，所以 RDD 同样可以提供完整的灾难恢复功能。

4. 两者互补

Hadoop 除了提供我们熟悉的 HDFS 分布式数据存储功能，还提供了 MapReduce 的数据处理功能、YARN 的资源调度系统。所以这里我们完全可以抛开 Spark，使用 Hadoop 自身的 MapReduce 来完成对数据的处理。

相反，Spark 也不是非要依附在 Hadoop 身上才能生存。但如上所述，毕竟它没有提供文件管理系统，所以，它必须和其他分布式文件系统进行集成才能运作。这里我们可以选择 Hadoop 的 HDFS，也可以选择其他的工具，比如 Red Hat GlusterFS。当需要外部的资源调度系统来支持时，Spark 可以跑在 YARN 上，也可以跑在 Apache Mesos 上（有关 Mesos 的内容会在下面的章节中详细讲述），当然也可以用 Standalone 模式。但 Spark 默认还是用在 Hadoop 上面的，毕竟，大家都认为它们的结合才是最好的。

4.4.3 Spark 2.0 的新特性

2016 年 7 月 26 日，Apache Spark 发布了 2.0 版本。Spark 2.0 正是基于过去两年获得的经验来构建的，它强化用户喜爱的功能，改善了让大家不满的地方。Spark 2.0 改进后更简单、更快速、更智能。

1. 更简单：标准的 SQL 及简化了的 API

Spark 让我们引以为豪的一点就是所创建的 API 简单、直观、便于使用，Spark 2.0 延续了这一传统，并在两个方面凸显了优势：

- 标准的 SQL 支持；
- 统一的 DataFrame/Dataset API。

在 SQL 方面，引入了新的 ANSI SQL 解析器，并支持子查询功能。Spark 2.0 可以运行所有 99 个 TPC-DS 查询（需要 SQL:2003 中的很多功能的支持）。由于 SQL 是 Spark 应用所使用的主要接口之一，所以对 SQL 功能的拓展大幅削减了将遗留应用移植到 Spark 时所需的工作。

在编程 API 方面，大大简化了 API。

- 在 Scala/Java 中统一了 DataFrame 与 Dataset: 从 Spark 2.0 开始，DataFrame 只是 Row 数据集的类型别名了。无论是 map、filter、groupByKey 之类的类型方法，还是 select、groupBy 之类的无类型方法，都可用于 Dataset 的类。此外，这个新加入的 Dataset 接口也被用作 Structured Streaming（结构化数据流）的抽象。由于 Python 和 R 语言不具备编译时类型安全的特性，所以 Dataset 的概念无法应用于这些语言的 API 中。相反，DataFrame 仍然是主要的接口，并且类似于这些语言的单节点的数据帧的概念。
- SparkSession: 这是一个新入口，取代了原本的 SQLContext 与 HiveContext。对于 DataFrame API 的用户来说，Spark 常见的混乱源头来自使用哪个“context”。现在你可以使用 SparkSession 了，它作为单个入口可以兼容两者。注意，原本的 SQLContext 与 HiveContext 仍然保留，以支持向下兼容。
- 更简单、性能更佳的 Accumulator API: 新的 Accumulator API 不但在类型层次上更简捷，同时专门支持基本类型。原本的 Accumulator API 已不再使用，但为了向下兼容仍然保留。
- 基于 DataFrame 的机器学习 API 将作为主 ML API 出现: 在 Spark 2.0 中，spark.ml 包及其“pipeline”API 会作为机器学习的主要 API 出现，尽管原本的 spark.mllib 包仍然保留，但以后的开发重点会集中在基于 DataFrame 的 API 上。
- 机器学习 pipeline 持久化: 在所有 Spark 所支持的语言里，用户都可以保存与载入机器学习的 pipeline 与模型。
- R 语言的分布式算法: 增加了在 R 语言中对 Generalized Linear Models（广义线性模型，GLM）、Naive Bayes（朴素贝叶斯算法）、Survival Regression（存活回归分析）与 K-Means（聚类算法）的支持。

2. 更快速：用 Spark 作为编译器

根据 2015 年对 Spark 的调查（调查报告见 <https://databricks.com/blog/2015/09/24/spark-survey-2015-results-are-now-available.html>），91% 的用户认为对 Spark 来说，性能是最为重要的。因此，性能优化一直是 Spark 2.0 规划的重点。

Spark 2.0 搭载了第二代 Tungsten 引擎，该引擎是根据现代编译器与 MPP 数据库的理念来构建的，它将这些理念用于数据处理中，其主要思想就是在运行时使用优化后的字节码，将整体查询合成单个函数，不再使用虚拟函数调用，而是利用 CPU 来注册中间数据。我们将这一技术称为“whole-stage code generation”。

表 4-1 列出了 Spark 1.6 与 Spark 2.0 在单核中处理单行数据所花费的时间（以十亿分之一秒为单位），证明了新一代 Tungsten 引擎的强大。Spark 1.6 使用 code generation 技术，这一技术如今在一些顶尖的商业数据库中在使用，正如我们看到的那样，在使用了新 whole-stage code generation 技术后，速度比之前快了一个数量级。

表 4-1 Spark 1.6 与 Spark 2.0 在单核中处理单行数据所花费的时间

原 语	Spark 1.6	Spark 2.0
filter	15ns	1.1ns
sum w/o group	14ns	0.9ns
sum w/ group	79ns	10.7ns
hash join	115ns	4.0ns
sort (8-bit entropy)	620ns	5.3ns
sort (64-bit entropy)	620ns	40ns
sort-merge join	750ns	700ns

图 4-5 对比了 Spark 1.6 与 Spark 2.0 在执行端对端 TPC-DS 查询时做的初步分析。

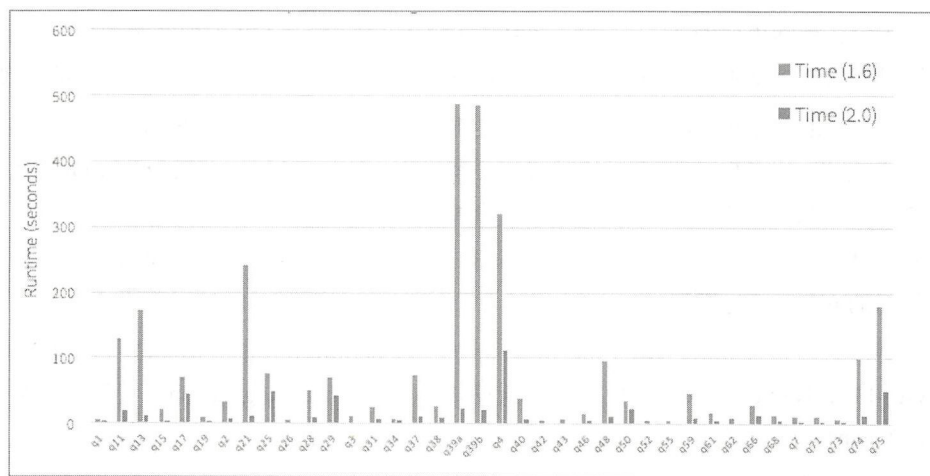


图 4-5 Spark 1.6 与 Spark 2.0 在执行端对端 TPC-DS 查询时做的初步分析

除此之外，为了改进 Catalyst optimizer 优化器对诸如 nullability propagation 之类常见查询的效果，还改进了矢量化 Parquet 解码器，使新解码器的吞吐量增加了三倍。

3. 更智能：Structured Streaming（结构化数据流）

作为首个尝试统一批处理与流处理计算的工具，Spark Streaming 一直是大数据处理的领导者。首个流处理 API 叫作 DStream，在 Spark 0.7 中初次引入，它为开发者提供了一些很强大的属性，包括只有一次语义、大规模容错及高吞吐量。

然而，在处理了数百个真实世界的 Spark Streaming 部署之后，我们发现需要在真实世界进行决策的应用经常需要不止一个流处理引擎。它们需要深度整合批处理堆栈与流处理堆栈，整合内部存储系统，并且要有处理业务逻辑变更的能力。因此，各大公司需要不止一个流处理引擎，并且需要能让它们开发端对端“持续化应用”的全栈系统。

Spark 2.0 的 Structured Streaming API 就是处理流数据的全新方式，相比于现有的流系统，Structured Streaming 进行了如下改进：

- 集成了 API 与批量作业。要运行流计算，开发人员只需针对 DataFrame/Dataset API 编写批量计算，Spark 会自动以流的方式运行计算。这种强大的设计意味着开发人员不必手工管理状态、故障或保持批量作业与应用同步。相反，数据流作业总是给出对同一数据批处理作业的相同答案。
- 与存储系统的事务交互。Structured Streaming 在处理容错性和一致性方面整体跨越引擎和存储系统，因此很容易编写更新的实时数据库应用，用于在存储系统之间可靠地移动数据。
- 可以与 Spark 其他部分丰富地集成。Structured Streaming 支持通过 Spark SQL 交互来查询数据流；使用 DataFrames，可以让开发人员构建完整的应用程序，而不仅仅是流 pipelines 静态数据。今后，希望有更多的与 MLlib 和其他库的整合。

作为这一愿景实现的第一步，Spark 2.0 搭载了初始版本的 Structured Streaming API，这是一个附在 DataFrame/Dataset API 上的（超小）扩展包。统一之后，对现有的 Spark 用户来说使用起来非常简单，他们能够利用在 Spark 批处理 API 方面的知识来回答实时的新问题。这里关键的功能包括支持基于事件时间的处理、无序/延迟数据、交互查询，以及与非流式数据源和汇的交互。

Spark 2.0 还更新了 Databricks 的工作区间用于支持 Structured Streaming，当执行流查询时，界面会自动展现状态，如图 4-6 所示。

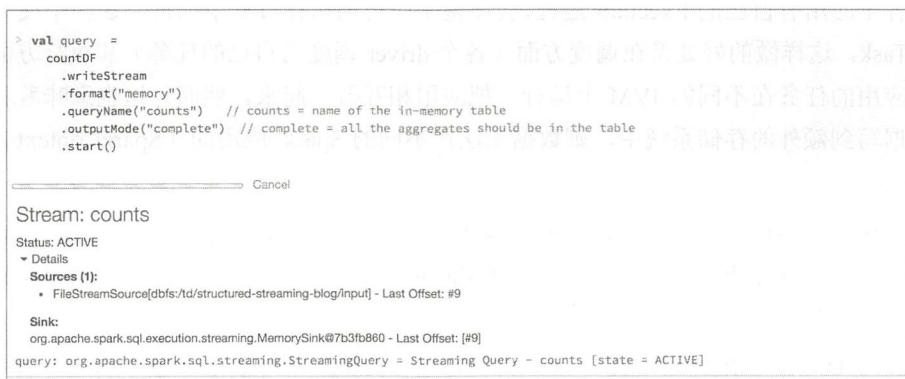


图 4-6 流查询时的状态

4.4.4 Spark 集群模式

下面介绍 Spark 如何在集群上运行。

1. 组件

Spark 应用在集群上以独立的进程集合运行,在主程序(称作 driver program)中以 SparkContext 对象来调节。

为了在集群上运行，SparkContext 可以与几个类型的 Cluster Manager（集群管理器）相连接，包括 Spark 自身的单独集群管理器或 Mesos、YARN，这些 Cluster Manager 可以在应用间分配资源。一旦连接，Spark 将获取在集群节点上的 Executor（执行器），也就是那些执行计算和存储应用数据的工作进程。然后，它将应用代码（以 JAR 或 Python 定义的文件并传送到 SparkContext）发送到 Executor。最后，SparkContext 发送 Task（任务）让 Executor 运行。

Spark 集群架构如图 4-7 所示。

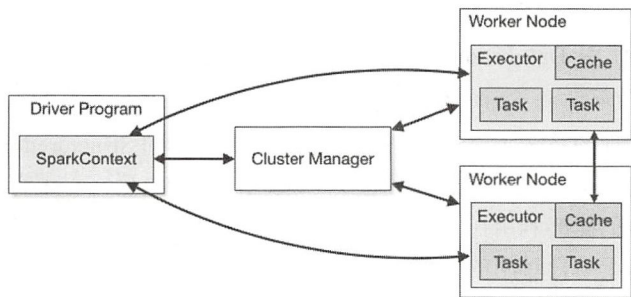


图 4-7 Spark 集群架构

关于这个架构有几个有用的地方需要注意:

- 各个应用有自己的 `Executor` 进程，会在整个应用的运行过程中保持并在多个线程中运行 `Task`。这样做的好处是在调度方面（各个 `driver` 调度它自己的任务）和执行方面（不同应用的任务在不同的 `JVM` 上运行）把应用相互孤立起来。然而，这也意味着若不把数据写到额外的存储系统中，则数据无法在不同的 `Spark` 应用间（`SparkContext` 的实例）共享。
- 潜在的 `Cluster Manager` 对于 `Spark` 来说是透明的。只要它能获取 `Executor` 的进程并在它们之间进行通信，就能在该 `Cluster Manager` 上支持其他应用（例如，`Mesos`、`YARN`），运行也相对简单。
- 因为 `Driver` 在集群上调度 `Task`，所以它运行的位置最好接近工作节点，在相同的局域网内就更好。如果你想对远程的集群发送请求，则较好的选择是为 `Driver` 打开一个 `RPC`。

让它就近提交操作而不是运行离工作节点很远的 driver。

2. Cluster Manager 类型

目前支持三类 Cluster Manager:

- Standalone——一种简单的集群管理，其包括一个很容易搭建集群的 Spark;
- Apache Mesos——一种通用的集群管理，可以运行 Hadoop MapReduce 和服务应用的模式;
- Hadoop YARN——Hadoop 2.0 中的资源管理模式。

Standalone 模式

启动 standalone master 服务器:

```
./sbin/start-master.sh
```

在启动正常后，master 会打印出 `spark://HOST:PORT UR`，以便 worker 来连接，或者作为“master”的参数传递给 `SparkContext`。也可以在 master 的 Web 界面看到这个 URL，默认是 `http://localhost:8080/`。

类似地，通过以下指令，你可以启动一个或多个 worker 来连接这个 master:

```
./sbin/start-slave.sh <master-spark-URL>
```

当 worker 启动后，就能在 master 的 Web 界面看到新增的节点列表，包括每个节点使用的 CPU、内存情况。

3. 提交应用

使用 `spark-submit` 脚本，可以让应用程序被提交给任何类型的集群。详见 <http://spark.apache.org/docs/latest/submitting-applications.html>。

4. 监控器

每个 Driver Program 都有一个 Web 界面，典型的是在 4040 端口，你可以看到有关运行的 Task、Executor 和存储空间大小等信息。你可以在浏览器中输入 `http://<driver-node>:4040` 来访问。关于监控器的更多选项请参阅 <http://spark.apache.org/docs/latest/monitoring.html>。

5. 任务调度

Spark 可以在应用间（在 Cluster Manager 水平）和应用里（如果在同一个 `SparkContext` 中有多个计算指令）进行资源分配。详见 <http://spark.apache.org/docs/latest/job-scheduling.html>。

6. 常用术语

集群概念中的常用术语如表 4-2 所示。

表 4-2 集群概念中的常用术语

术 语	含 义
Application（应用）	在 Spark 上构建的程序，在集群上由 Driver Program 和 Executor 组成
Application jar	包含了 Spark 应用的 jar 包。用户的 jar 包不应该包含 Hadoop 或 Spark 的包，而应该在运行时才进行添加
Driver Program（驱动程序）	运行 main()函数的进程，同时创建 SparkContext
Cluster Manager（集群管理器）	在集群上获取资源的扩展服务，比如 Standalone 管理器、Mesos、YARN
Deploy Mode（部署节点）	用来区分 Driver 进程的运行位置。在“cluster”模式中，框架在集群内部启动 Driver，而在“client”模式中，提交者在集群外部启动 Driver
Worker Node（工作节点）	任何在集群中可以运行应用的节点
Executor（执行器）	在 Worker Node 中为应用启动的一个进程，它可以运行 Task，并在内存或硬盘中保存数据。每一个应用都有属于自己的 Executor
Task（任务）	一个可以发送给 Executor 执行的工作单元
Job（作业）	一个由多任务组成的并行计算，并能从 Spark action 中获得回应(比如 save、collect)；你可以在 Driver 的日志中看到这项内容
Stage（阶段）	每个 Job 被分为很多小的 Task 集合，这个进程被称为 Stage（与 MapReduce 中的 map 和 reduce 阶段相似）；你可以在 Driver 的日志中看到这项内容

4.5 Mesos

物理机和虚拟机是数据中心的典型的计算单元。在应用部署后，这些机器需要安装各种配置工具来管理这些应用。机器通常被组织成集群来提供独立的服务，系统管理员则监督其日常的运作。当这些集群达到其最大容量时，需要多机联网来处理负载，这就给集群的扩展带来了挑战。

2010 年，UC Berkeley 大学就对上述问题提出了解决方案，这就是现在的 Apache Mesos（后简称 Mesos）。Mesos 抽象了 CPU、内存、硬盘资源，让数据中心的对外功能就像一个大的机器。Mesos 创建了一个单独的底层集群来提供应用程序所需要的资源，而不会超出虚拟机和操作系统的性能限制。

4.5.1 Mesos 简介

Mesos 是 Apache 下的开源分布式资源管理框架，它被称为分布式系统的内核，使用内置 Linux 内核相同的原理，只是在不同的抽象层次。该 Mesos 内核运行在每个机器上，在整个数据中心和云环境内向应用程序（例如，Hadoop、Spark、Kafka、Elasticsearch 等）提供资源管理和资源负载的 API 接口。

1. Mesos 架构

图 4-8 是 Mesos 的架构组成。

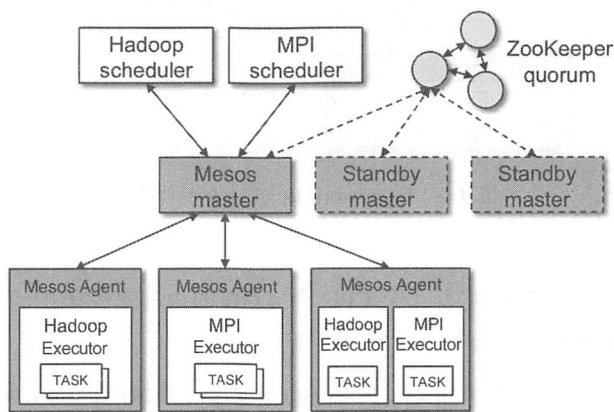


图 4-8 Mesos 的架构组成

Mesos 主要由 master 守护线程和 Mesos framework 组成。master 守护线程用于管理运行在每个集群节点的 Agent 守护线程。Mesos framework 用来运行 Agent 上面的 Task。

master 通过 framework 来使资源成为 resource offer，这样就能使用细颗粒度的资源共享（CPU、RAM 等）了。每个 resource offer 都包含<agent ID, resource1: amount1, resource2: amount2, ...>这样的列表。master 根据给定的组织策略来决定多少资源能够提供给每个 framework，其中的策略有 fair sharing（公平共享）和 strict priority（严格优先级）。为了支持策略的多样化，master 采用了模块化架构，可以很容易地通过一个插件机制来增加新的分配模块。

framework 运行在 Mesos 的顶层，主要由两个组件组成：scheduler 和 Executor。其中，scheduler 注册到 master 来分配资源，而 Executor 进程则启动 Agent 节点来运行 framework 任务。同时，master 决定将多少资源提供给每个 framework，scheduler 决定使用哪些资源。framework 在接收了资源后会传递给 Mesos 一个描述符，告诉 Mesos 想要启动哪个 Task。接着，Mesos 启动对应 Agent 的 Task。

2. resource offer 的示例

图 4-9 显示了 framework 运行 Task 的流程。

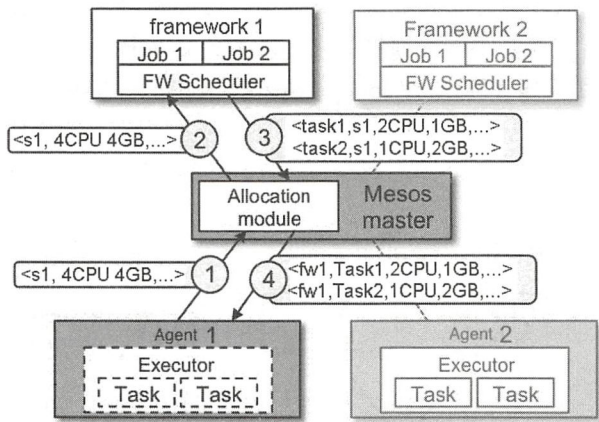


图 4-9 framework 运行 Task 的流程

整个流程描述如下：

- （1）Agent 1 报告给 master 它有 4 个 CPU 和 4 GB 的内存是空闲的。然后，master 就会调用分配策略模块，告诉 framework 1 应分配掉所有可用资源。
- （2）master 发送一个 resource offer，描述 Agent 的哪些资源对 framework 1 来说是可用的。
- （3）framework 的 scheduler 回复给 master 运行在 Agent 的两个 task 信息，第一个 task 使用<2 CPUs,1 GB RAM>，第二个 Task 使用<1 CPUs,2 GB RAM>。
- （4）master 发送 Task 给 Agent，它分配适当的资源 framework 的 Executor，从而启动了两个 task（图中以虚线边框表示）。由于 1 个 CPU 和 1 GB 内存仍然未分配，所以分配模块现在可以提供它们到 framework 2。

此外，当 Task 完成和新的资源变成空闲时，这一 resource offer 过程会重复执行。

4.5.2 设计高可用的 Mesos framework

Mesos framework 用于管理 Task。为了使 Mesos framework 高可用，必须持续在各种故障场景中正确地管理 Task。开发人员需考虑以下常见的故障条件：

- framework scheduler 所连接的 Mesos master 出现故障，例如，崩溃或失去了网络连接。如果 master 已经配置为 high-availability mode（高可用性模式），则推动另一个 Mesos master 副本成为当前的 leader。在这种情况下，scheduler 应与新的 master 重新注册，并

确保 Task 的状态是一致的。

- framework scheduler 所运行的主机出现故障。为了确保该 framework 仍然可用，并能继续安排新的 Task，framework 开发者应确保在不同的节点上运行 scheduler 的多个副本，并且在前任 leader 失败后，该备份副本能晋升为当前新的 leader。Mesos 本身并没有规定 framework 开发者应该如何处理这种情况。一种不错的方式是，部署多个使用长时间运行的 Task 的 framework scheduler 的副本，例如 Apache Aurora 或 Marathon。
- Task 所运行的主机出现故障。可替代的节点本身可能没有故障，但该节点上的 Mesos Agent 可能无法与 Mesos master 通信，例如，由于进行了网络分区所以无法通信。

需要注意的是，这些故障可能会同时发生。

1. Mesos 现有设计

以下是影响 Mesos 高可用性方面的设计。

- Mesos 默认提供组件之间的不可靠通信：消息传递是“at-most-once（最多一次）”，在这个过程中，消息有可能被丢弃。framework 开发者应该预料到发送的信息可能未被收到，而应采取相应的纠正措施。要检测一个消息是否丢失，framework 通常使用超时机制。例如，如果一个 framework 尝试启动一个 Task，则该消息可能不被 Mesos master 所接收（例如，由于瞬时网络故障）。为了解决这个问题，该 framework scheduler 在试图启动一个新的 Task 后，设置超时。如果在超时后，scheduler 还没有看到新 task 的状态更新，应该采取纠正措施了，例如，通过执行 task state reconciliation（任务状态和解，详情可以参考 <http://mesos.apache.org/documentation/latest/reconciliation/>），并在必要时再启动 Task 的新副本。
 - 一般情况下，分布式系统不能区分出“遗失”的消息和延迟的消息。在上面的例子中，scheduler 可能会在超时后立即看到第一个 Task 开启状态更新了，而此时，它已经开始采取纠正措施了。
 - Mesos 实际上提供了任意进程对之间的有序的（但不可靠的）消息传递。比如，如果一个 framework 发送消息 M1 和 M2 到 master，则 master 可能会收不到任何消息，或者只收到 M1，或者只收到 M2，或先收到 M1 后收到 M2，但它不会先收到 M2 后收到 M1。
 - 为了方便 framework 开发者，Mesos 提供了 Task 状态更新的可靠传输。Agent 将 Task 状态更新到磁盘上，然后将它们转发给 master。master 发送状态更新到适当的 framework scheduler。当 scheduler 确认状态更新了，master 转发该确认并返回 agent，这样所存储的状态就能被标记为可被垃圾回收。如果 Agent 未在一定的时间内接收

到 Task 状态更新的确认，则它会反复重新发送状态更新到 master，将更新再次转发给 scheduler。因此，在 Agent 和 scheduler 都保持可用的情况下，Task 状态更新将交付“at least once（至少一次）”。为了处理 Task 状态更新可能被传递多于一次的这种情况，应该让 framework 的逻辑处理成幂等。

- Mesos master 在内存中存储活动 Task 和注册 framework 的信息，它并不把信息保存到磁盘或试图在 master 故障后将信息转移后保存。这样做有助于 Mesos master 扩展到拥有更多 Task 和 framework 的大型集群。这种设计的缺点是，在发生故障后，需要做更多努力来恢复丢失的内存 master 状态。
- 如果所有的 Mesos master 都不可用（例如，崩溃或无法访问），则集群应继续运作，现有的 Mesos agent 和用户 Task 应继续运行。然而，新的 Task 无法安排，并且 framework 不会收到关于之前启动的 Task 的 resource offer 或状态更新。
- Mesos 没有规定 framework 应如何实现及如何处理失效的情况。相反，Mesos 尝试为 framework 开发人员提供他们需要实现这一行为本身的工具。不同的 framework 可能会选择不同的方式来处理故障，这样取决于具体需求。

2. 对于设计高可用 framework 的建议

高可用的 framework 设计通常应遵循一些共同的模式：

（1）为了能容忍 scheduler 故障，framework 应运行多个 scheduler 实例（典型值是三个实例）。在任何给定的时间，只有其中一个实例是 leader：此实例连接到 Mesos master，接收的 resource offer 和 Task 状态更新并启动新的 Task。其他 scheduler 副本是 follower：只有当 leader 发生故障了，follower 之一才会被选为新的 leader。

（2）scheduler 需要一个机制来决定何时在当前 leader 故障时，选出新的 leader。一般使用 Apache ZooKeeper 或 etcd 来协调服务实现。

（3）在选出一位新的 scheduler leader 后，新 leader 应重新连接 Mesos master。在与 master 注册后，该 framework 应设置其 FrameworkInfo 的 id 字段为所分配的故障 scheduler 实例的 ID。这确保了 master 认识到，该连接不会重新启动会话，而是继续使用之前故障的 scheduler 实例会话（注：在旧 scheduler leader 从 master 断开连接后，在默认情况下，master 将立即杀死所有与故障 framework 关联的 Task 及 Executor。而这种行为对于典型的生产环境是非常不可取的！为了避免出现这种情况，在设计高度可用的 framework 时，应将其 FrameworkInfo failover_timeout 字段值设置得相对宽泛点。为了避免在生产环境中 Task 被意外破坏，许多 framework 会使用 1 周以上的 failover_timeout）。

- 在目前的实现中，master 故障过程中不会保留 framework 的 failover_timeout。因此，如

果一个 framework 失效了，在 failover_timeout 到达前，如果处于领导地位的 master 也故障了，那么新当选的处于领导地位的 master 并不会知道该 framework Task 应在一段时间后需要被杀死。因此，如果 framework 不重新注册，则这些 Task 将继续无限期运行，但这些 Task 是会被孤立的。这种行为可能会在 Mesos 的后续版本中修正。有关这个 bug 的描述，可以参见 MESOS-4659: <https://issues.apache.org/jira/browse/MESOS-4659>。

(4) 在连接到 Mesos master 之后，新的处于领导地位的 scheduler 应确保其本地状态与集群的当前状态保持一致。例如，假设先前处于领导地位的 scheduler 尝试启动一个新的 Task，然后立即就失败了。该 Task 可能已成功启动了，在这点上新当选的 leader 将开始收到有关其状态的更新信息。为了处理这种情况，framework 通常使用强一致性的分布式数据存储来记录处于活动和尚未完成的 Task 的信息。实际上，有很多协调服务可用于这类 leader 选举，比如 ZooKeeper 或 etcd。一些 Mesos framework 如 Apache Aurora 等，使用 Mesos replicated log 的方式来实现此目的（有关 replicated log 的详细内容，可以参阅 <http://mesos.apache.org/documentation/latest/replicated-log-internals/>）。

- 数据存储应该记录 scheduler 在采取动作前的意图。举例来说，当 scheduler 想启动一个 Task 时，它需要先在数据存储里面记录下这个动作的意图，接着才发动“启动 Task”的消息给 Mesos master。如果该 scheduler 的实例故障了，另外一个新的 scheduler 被选为了 leader，那么这个新的 leader 能从数据存储中找到所有可能在集群中运行的 Task。这种模式被称为 write-ahead logging（预写日志，简称 WAL，详情可参考 https://en.wikipedia.org/wiki/Write-ahead_logging），往往使用数据库系统和文件系统来改善可靠性。下面的内容需要注意：
 - 首先，scheduler 在启动 Task 前，必须存储其意图。若 Task 已经启动，但 scheduler 在存储意图前发生故障，那么新的处于领导地位的 scheduler 将无法获知这个新的 Task 的信息。如果发生这种情况，则新的 scheduler 实例将在对该 Task 一无所知的情况下就开始接收 task 的状态更新，这并不是一个好方法。
 - 其次，scheduler 应确保其意图在启动 Task 的过程中能持续记录在数据存储中（例如，它应该等待所有的副本在数据存储中已经达到法定数目，以确认可以执行写操作）。

3. Task 的生命周期

Mesos Task 会经历一系列的状态。一个 Task 的当前状态判断依据是 Agent 在哪个 Task 上运行。framework scheduler 通过与 Mesos master 的交互来获知 Task 的当前状态。具体来说，是通过监听 Task 状态更新及执行任务 Task 状态和解来实现的。

framework 可以使用 state machine（状态机）来表示 Task 的状态，其中包括一个初始状态和几种可能的终止状态：

- Task 启动时是 TASK_STAGING 状态。当 master 接收到 framework 启动 Task 的请求而 Task 还没有开始运行时，Task 就为此状态。在该状态，Task 获取到了它的依赖关系。比如，使用 Mesos fetcher cache（详情见 <http://mesos.apache.org/documentation/latest/fetcher/>）。
- 该 TASK_STAGING 状态是可选的，主要供自定义的 Executor 使用。它可以用来描述一个事实，即自定义的 Executor 已经了解 Task（也许开始获取其依赖关系），但尚未开始运行。
- 在 Task 成功运行后，其状态将转为 TASK_RUNNING。如果 Task 启动失败，则状态转为以下终止状态之一。
- 如果一个 framework 尝试启动一个 Task，但在超时后还没有接收到状态更新，那么 framework 应执行 reconciliation（和解）。也就是说，它应该咨询 master 当前 Task 的状态。如果是未知的 Task，则 master 回复 TASK_LOST 状态更新。然后，framework 可以利用这一点来判断哪些 Task 是慢启动的，哪些是 master 所不知道的。注意，这种技术的正确性取决于 scheduler 和 master 之间的通信是否有序。
- TASK_KILLING 状态是可选的，旨在表明 Executor 已经接收到了杀掉 Task 的请求，但是实际上 Task 还没有被杀死。对于需要一段时间才能正常终止 Task 的情况来说，这非常有用。除非该 framework 有 TASK_KILLING_STATE 的能力，否则 Executor 不能生成这种状态。

有以下几种终止状态：

- 当一个 Task 成功完成后，使用 TASK_FINISHED。
- TASK_FAILED 表示 Task 有错误中止。
- TASK_KILLED 表示 Task 被 Executor 杀掉了。
- TASK_LOST 表示该 Task 所运行的 Agent 已经与当前 master（通常是由于网络分区或 agent 主机故障）失去了联系。
- TASK_ERROR 表示由于 Task 错误，Task 尝试启动失败。

注意，相同的 Task 状态可以在几个不同的情况下使用（但通常是相关联的）。

4. 处理分区或失效的 Agent

该 Mesos master 跟踪注册 Agent 的可用性和健康状况，使用的是两种不同的机制：

- （1）master 与 Agent 之间持久的 TCP 连接的状态。

(2) 定期使用 ping 消息给 Agent 来做 health checks (健康检查)。在超时时间内, 若 master 发送 “ping” 消息给 Agent, 则能收到 “ping” 的响应消息。如果它不及时对连续的一定数量的 ping 消息做出响应, 则可以判断 agent 已经发生故障了。这种行为是由 master 的 `--agent_ping_timeout` 和 `--max_agent_ping_timeouts` 标识控制的。

如果连接到 Agent 的持久的 TCP 连接出现中断或 Agent 的 health checks 失败, 则 master 判断 Agent 发生故障, 并采取措施将其从集群中移除。

- 如果 TCP 连接出现中断, 则 Agent 被认为断开了连接。
- 如果 framework 在 checkpointing (检查点), 则不采取立即行动。该 Agent 有机会重新连接, 直到 health checks 超时。
- 如果 framework 没在 checkpointing, 则所有的 framework 的 Task 和 Executor 都被认为丢失了。master 立即发送这个 task 的 TASK_LOST 状态更新。这些更新不能可靠地传递给 scheduler。该 Agent 有机会重新连接, 直到 health checks 超时。如果 Agent 重新连接, 则之前发送的 TASK_LOST 状态更新的 Task 就会被杀掉。
- 针对此行为的基本原理是, 采用标准的 TCP 设置, 在 master 和 Agent 之间持续的 TCP 连接的错误对于网络分区来说, 更可能对应于 Agent 的错误 (例如, mesos-agent 意外终止), 因为 Mesos health-check 超时比相应的 TCP 超时的典型值小得多。由于非 checkpointing 的 framework 将无法使 mesos-agent 进程重新启动, master 发送 TASK_LOST 状态更新, 使这些 Task 可以及时重新调度。当然, 该 TCP 差错不对应于网络分区的情况, 在某些环境中并不适用。
- 如果 Agent 的 health checks 失败了, 则它将被纳入移除计划。该移除的比例可以通过 master 来设置 (见 master 标识 `--agent_removal_rate_limit`), 以避免同时移除大量的 Agent (例如, 在网络分区过程中)。
- 当清除 Agent 的时间来到时, master 从已注册 Agent 的列表中移除 Agent。master 将发送一个 slaveLost 回调给每个注册的 scheduler driver; 它也发送 TASK_LOST 状态更新给每个被移除了的 Agent。注意, 回调和 Task 状态更新都是由 master 进行传递的。例如, 如果 master 或 scheduler 故障, 或者在这些消息的传递过程中网络连接出现问题, 则它们将不会重新发送。
- 同时, 在被移除的 Agent 中的 Task 将继续运行, Agent 将反复尝试重新连接 master。一旦被移除的 Agent 重新连接 master (例如, 网络分区已恢复), 重新尝试注册将被拒绝, Agent 将被要求关闭。接着, Agent 将关闭所有正在运行的 Task 和 Executor。被移除的 Agent 中的 persistent volumes (持久化卷) 和 dynamic reservations (动态保留) 将被保留。

- 被移除的 Agent 可以通过重启 mesos-agent 进程来重新加入集群。当被移除的 Agent 被 master 关闭时，Mesos 确保在下次 mesos-agent 启动时（使用相同的主机，相同的工作目录），这个 Agent 将会接收到新的 Agent ID，注意该 Agent 就能被当成一个新加入的 Agent。该 agent 就能重新获得之前创建的 persistent volumes 和 dynamic reservations，即使该 agent ID 所关联的资源已经改变了。

通常情况下，framework 响应失败或分区的 agent 是通过调度在丢失的 Agent 上运行的 Task 的新副本来实现的。但需要注意，丢失的 Agent 有可能还存活，但被 master 划分了分区，以至于无法与它通信。根据网络分区的性质，Agent 的 Task 可能仍然能够与集群中的外部客户机或其他主机进行通信。framework 可以采取措​​施来阻止这样的情况发生（例如，Task 连接到 ZooKeeper，如果 ZooKeeper 的 session 过期就停止操作），但 Mesos 把这些处理的细节交给了 framework 开发者自己处理。

5. 处理分区或失效的 master

上述描述的行为对于新的 Mesos master 立即选出的期间并不适用。如上所述，Mesos master 状态仅保留在内存中，因此，当处于领导地位的 master 故障，一个新的 master 被选出，而新的 master 对于集群目前的状态知之甚少。相反，随着 framework 和 Agent 注意到一个新的 master 当选，通过 reregister（重新注册）来使它重建此类状态信息。

（1）framework 注册。

当 master 出现故障时，被连接到之前处于领导地位的 master 的 framework 应重新连接新的处于领导地位的 master。MesosSchedulerDriver 用于处理大多数检测的细节，比如之前处于领导地位的 master 已故障，需要连接新的处于领导地位的 master；当 framework 已经成功地注册到新的处于领导地位的 master 时，该 reregisteredscheduler driver 回调将会被调用。

（2）Agent 注册。

新的 master 已当选之后但在一个给定的 Agent 已经重新注册或 agent_reregister_timeout 已经起效之前，在此期间，试图调和 Agent 上运行的 Task 的状态将不会返回任何信息（因为 master 无法准确判断 Task 的状态）。

如果在未超时的情况下（由 --agent_reregister_timeout 配置标记控制），agent 未重新注册到新的 master，master 将会标记该 Agent 是失效的，并遵循上述相同的步骤。但是，有一点不同，在默认情况下，Agent 被允许重新连接到故障转移的 master，即便是 agent_reregister_timeout 已经启用了还是一样。这意味着 framework 可能会看到一个 TASK_LOST 更新的 Task，但后来发现，该 Task 已经在运行了（因为正在运行的 Task 的 agent 被允许重连）。



4.6 实战：基于 Spark 的词频统计

下面将演示如何基于 Spark 框架实现词频统计功能。

4.6.1 项目概述

下面创建一个名为“spark-word-count”的应用。在该应用中，使用 Spark 实现对文章中单词的出现频率进行统计。

为了能够正常运行该应用，需要在应用中添加如下 Spark 依赖：

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <spark.version>2.3.0</spark.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.11</artifactId>
        <version>${spark.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
        <version>${spark.version}</version>
    </dependency>
</dependencies>
```

4.6.2 项目配置

我们事先在 D 盘下准备一个 txt 文本文件——rfc7230.txt。该文件是 HTTP 规范 RFC 7230 的全文内容。

当应用启动之后，会读取该文件的内容，作为词频统计的基础。



4.6.3 编码实现

以下是应用 JavaWordCount 的所有内容：

```
package com.waylau.spark;
import scala.Tuple2;

import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.sql.Session;

import java.util.Arrays;
import java.util.List;
import java.util.regex.Pattern;

public final class JavaWordCount {
    private static final Pattern SPACE = Pattern.compile(" ");

    public static void main(String[] args) throws Exception {

        if (args.length < 1) {
            System.err.println("Usage: JavaWordCount <file>");
            System.exit(1);
        }

        SparkSession spark = SparkSession.builder().appName(
            "JavaWordCount").getOrCreate();

        JavaRDD<String> lines = spark.read().textFile(args[0]).javaRDD();

        JavaRDD<String> words = lines.flatMap(s -> Arrays.asList(
            SPACE.split(s)).iterator());

        JavaPairRDD<String, Integer> ones = words.mapToPair(s -> new Tuple2<>
            (s, 1));

        JavaPairRDD<String, Integer> counts = ones.reduceByKey((i1, i2) ->
            i1 + i2);
```



```
List
```

4.6.4 运行

为了能够正常运行该程序，我们要在应用启动参数中指定待统计的文件 rfc7230.txt 所在的路径。同时设置程序为 local 模式。设置启动参数，如图 4-10 所示。

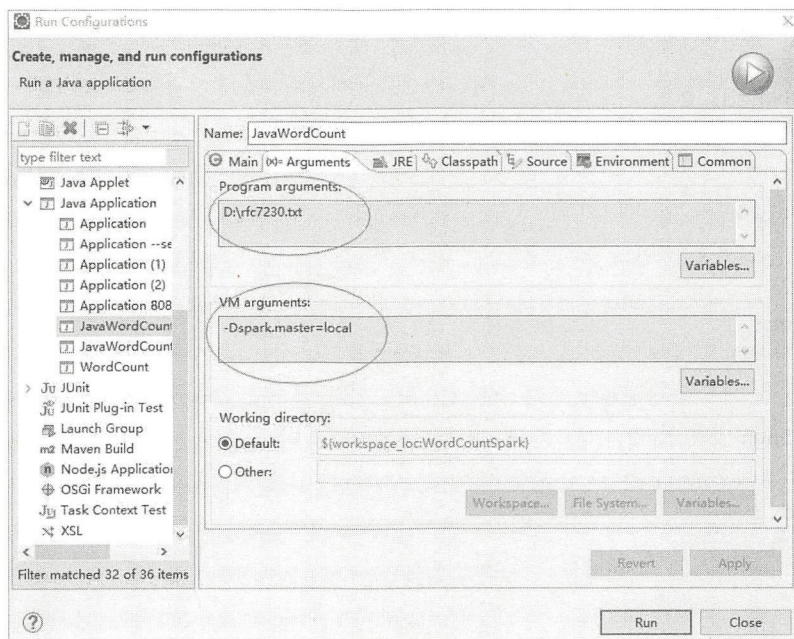


图 4-10 设置启动参数

在应用正常启动之后，应能在控制台看到如下词频统计信息：

```
Unfortunately,: 2
.....56: 1
constraints: 1
retry.: 2
Saurabh: 1
```



```
"accelerator": 1
desirable: 1
listening: 5
components.: 1
GmbH: 1
order: 29
7234,: 1
Compression: 2
Supported): 1
behind: 2
merge: 1
end: 6
been: 64
evaluating: 1
Failures: 2
accomplished: 2
"?": 8
A.2.: 2
clients: 18
9.: 2
knows: 2
selective: 1
less: 2
Reed,: 1
supporting: 2
64]: 1
expanded.: 1
Nathan: 1
RWS: 12
ignore: 13
entry: 2
(DQUOTE: 1
are: 145
"path-abempty",: 1
2.: 5
Nilsson,: 1
Isomaki,: 1
Content-Type:: 1
```




```
consists: 4
undesirable: 1
Miles: 1
qvalues: 1
records: 1
different: 11
Smuggling: 2
trailer-part: 5
necessitated: 1
...
```

当然，词频统计列表较长，这里只展示了列表中的部分单词。

上面例子中的代码可以在 <https://github.com/waylau/distributed-systems-technologies-and-cases-analysis> 的 spark-word-count 程序中找到。

5 chapter

第 5 章 分布式存储





5.1 分布式存储概述

互联网每天都产生数以亿计的数据，如何正确存储、解析、利用这些数据，是每个数据公司所面临的挑战。传统的关系型数据库已不能应对大规模的数据处理，NoSQL 应运而生。

NoSQL 泛指非关系型数据库，旨在应对大规模数据集合的多重数据种类所带来的挑战，尤其是大数据应用的难题。

5.1.1 使用场景

传统的关系型数据库在部署上往往采用单机部署的方式，这在容错性方面存在限制。而且关系性数据库在大数据处理方面能力较弱，不适合海量数据的应用场景。

以 NoSQL 为代表的分布式存储正着力于解决上述问题。以下场景非常适合使用 NoSQL：

- 分布式部署。主流的 NoSQL 都支持分布式存储，这非常适合对容错性要求比较高的业务场景。
- 海量数据存储。当数据量达到 TB 规模以上时，无论是 MySQL 还是 Oracle，传统的关系型数据库都已经无法支撑数据的及时处理，此时宜选用 NoSQL。
- 高性能。分布式存储产品，充分利用多处理器和多核计算机的性能，并考虑在分布于多个数据中心的大量这类服务器上运行。它可以一致且无缝地扩展到数百台机器，因此，即便在高负载的场景下，依然拥有良好表现。

是否使用分布式存储，需要根据自己的项目情况来斟酌：一是要看自己应用的数据量规模是否达到了 TB 级别以上；二是要看业务对于应用的容错、可扩展性、性能等方面的考量。同时，使用分布式存储相比于传统的关系型数据库，需要一定的学习成本，所以在进行技术选型时，也需要综合考虑企业自身的人力资源情况。

5.1.2 常用技术

分布式存储技术在业界已经非常成熟。Bigtable 在 Google 自己的产品和项目上有着广泛的应用。Apache HBase、Apache Cassandra 都是开源的分布式存储技术，可以实现与 Apache Hadoop、Apache Spark 等计算平台的无缝集成。Memcached、Redis 是常用的分布式缓存方案，适用于高性能的缓存数据存取。MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库中功能最丰富、最像关系数据库的，因此理论上可以直接作为关系型数据库的替代品。





5.2 Bigtable

Bigtable 是非关系型数据库，是一个稀疏的、分布式的、持久化存储的多维度排序 map，用于快速且可靠地处理 PB 级别的数据，并且将其部署到上千台机器上。

Bigtable 已经实现了以下几个目标：适用性广泛、可扩展、高性能和高可用性。

5.2.1 Bigtable 的数据模型

map 在编程语言中是一种非常常见的数据结构，由 key 和 value 组成。map 的索引是由 row key（行键）、column key（列键）及 timestamp（时间戳）来确定的；map 中的每个 value 都是一个未经解析的 byte 数组。映射关系如下：

```
(row:string,column:string,time:int64)->string
```

为了更加形象地解释 Bigtable 的数据模型，我们先举个具体的例子，从这个例子可以看出当初 Bigtable 的设计者是如何设计这个模型的。假设我们想要存储海量的网页及相关信息，我们姑且称这个特殊的表为 Webtable。在 Webtable 里，我们使用 URL 作为 row key，使用网页的某些属性作为 column 名，将网页的内容存在“contents:”列中，并用获取该网页的 timestamp 作为标识，如图 5-1 所示。

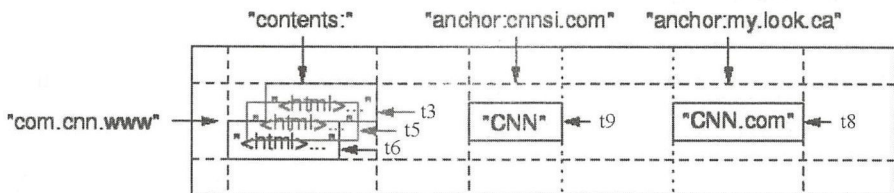


图 5-1 Webtable 示例

图 5-1 是一个 Webtable 的部分片段，用于存储 Web 页面。row 名是一个反向 URL。contents column 族存放的是网页的内容，anchor column 族存放引用该网页的锚链接文本。CNN 的主页被 Sports Illustrated 和 MY-look 的主页引用，因此该 row 包含了名为“anchor:cnnsi.com”和“anchor:my.look.ca”的 column。每个 anchor 链接只有一个版本（时间戳标识了列的版本，t9 和 t8 分别标识了两个 anchor 链接的版本）；contents column 则有三个版本，分别由时间戳 t3、t5 和 t6 标识。

1. Row（行）

表中的 row key 可以是任意字符串（目前支持最大 64KB 的字符串，但是对大多数用户，10~100 个字节就足够了）。对同一个 row key 的读或者写操作都是原子的（不管读或者写这一





row 里有多少个不同 column)，这个设计决策能够使用户很容易地理解程序在对同一个 row 进行并发更新操作时的行为。

2. Column Family (列族)

column key 组成的集合叫作“column family (列族)”，column family 是访问控制的基本单位。存放在同一 column family 下的所有数据通常都属于同一个类型（我们可以把同一个 column family 下的数据压缩在一起）。column family 在使用之前必须先创建，然后才能在 column family 中的任何 column key 下存放数据；在 column family 创建后，其中的任何一个 column key 下都可以存放数据。一张表中的 column family 不能太多（最多几百个），并且 column family 在运行期间很少改变。与之相对应地，一张表可以有无限多个 column。

column key 的命名语法如下：family:qualifier。column family 的名字必须是可打印的字符串，而限定符的名字可以是任意字符串。比如，Webtable 有个 column family 叫 language，language 用来存放撰写网页的语言。我们在 language 中只使用一个 column key，用来存放每个网页的语言标识 ID。Webtable 中另一个有用的 column family 是 anchor。这个 column family 的每一个 column key 代表一个 anchor 链接，如图 5-1 所示。这个 column family 的 qualifier（限定符）是引用该网页的站点名；这个 column family 的每列数据项存放的是链接文本。

访问控制、磁盘和内存的使用统计都是在 column family 层面进行的。在 Webtable 的例子中，上述控制权限能帮助我们管理不同类型的应用：我们允许一些应用添加新的基本数据，一些应用可以读取基本数据并创建继承的 column family，一些应用则只允许浏览数据（甚至可能因为隐私的原因不能浏览所有数据）。

3. Timestamp (时间戳)

在 Bigtable 中，表的每一个数据项都可以包含同一份数据的不同版本，不同版本的数据通过 timestamp 来索引。Bigtable 的 timestamp 类型是 64 位整型。Bigtable 可以给 timestamp 赋值，用来表示精确到毫秒的“实时”时间。用户程序也可以给 timestamp 赋值。如果应用程序需要避免数据版本冲突，那么它必须自己生成具有唯一性的 timestamp。在数据项中，不同版本的数据按照 timestamp 倒序排序，即最新的数据排在最前面。

为了减轻多个版本数据的管理负担，每一个 column family 都配有两个设置参数，Bigtable 通过这两个参数可以对废弃版本的数据自动进行垃圾收集。用户可以指定只保存最后 n 个版本的数据，或者只保存“足够新”的版本的数据，比如，只保存最近 7 天写入的数据。

在 Webtable 的例子中，contents:列存储的时间信息是网络爬虫抓取一个页面的时间。上面提及的垃圾收集机制可以让我们只保留最近三个版本的网页数据。





5.2.2 Bigtable 的实现

Bigtable 包括三个主要组件：链接到客户程序中的库、一个 master 服务器和多个 tablet 服务器。针对系统工作负载的变化情况，Bigtable 可以动态地向集群中添加或者删除 tablet 服务器。

master 服务器主要负责以下工作：为 tablet 服务器分配 tablet，检测新加入的或者过期失效的 tablet 服务器，对 tablet 服务器进行负载均衡，以及对保存在 GFS 上的文件进行垃圾收集。它还处理对模式的相关修改操作，例如建立表和 column family。

每个 tablet 服务器都管理一个 tablet 的集合（通常每个服务器有大约数十个至上千个 tablet）。每个 tablet 服务器负责处理它所加载的 tablet 的读写操作，以及在 tablet 过大时对其进行分割。

和很多 single-master 类型的分布式存储系统类似，客户端读取的数据都不经过 master 服务器，而是客户端程序直接和 tablet 服务器通信进行读写操作。由于 Bigtable 的客户端程序不必通过 master 服务器来获取 tablet 的位置信息，因此，大多数客户程序甚至完全不需要和 master 服务器通信。在实际应用中，master 服务器的负载是很轻的。

一个 Bigtable 集群存储了很多表，每个表都包含一个 tablet 的集合，而每个 tablet 都包含某个范围内的 row 的所有相关数据。在初始状态下，一个表只有一个 tablet。随着表中数据的增加，它被自动分割成多个 tablet，在默认情况下，每个 tablet 大约为 100~200 MB。

1. tablet 的位置信息

tablet 的位置信息是使用一个三层的、类似 B+树的结构存储，如图 5-2 所示。

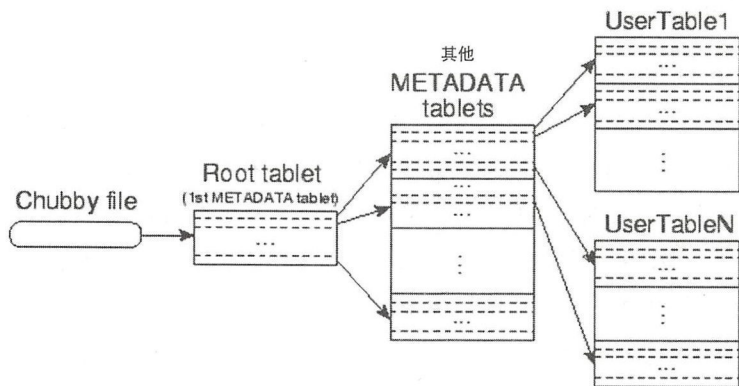


图 5-2 tablet 的位置信息

第一层是一个存储在 Chubby 中的文件，它包含 root tablet 的位置信息。Chubby 是 Google 公司设计的一种分布式系统的锁服务，其目的是提供粗粒的锁定及可靠的存储。本书不会对 Chubby 进行详细讲解，若读者有兴趣，可以参阅 Mike Burrows 的论文 *The Chubby lock service for*



loosely-coupled distributed systems。root tablet 包含一个特殊的 METADATA 表，用于记录所有的 tablet 位置信息。METADATA 表的每个 tablet 都包含一个用户 tablet 的集合。root tablet 实际上是 METADATA 表的第一个 tablet，只不过对它的处理比较特殊——root tablet 永远不会被分割，这就保证了 tablet 的位置信息存储结构不会超过三层。

在 METADATA 表里面，每个 tablet 的位置信息都存放在一个 row key 下面，而这个 row key 是由 tablet 所在的表的 identifier 和 tablet 的最后一行编码而成的。METADATA 的每一行都存储了大约 1KB 的内存数据。在一个大小适中的、容量限制为 128MB 的 METADATA 的 tablet 中，采用这种三层结构的存储模式，可以标识 2^{34} 个 tablet 的地址（如果每个 tablet 存储 128MB 数据，那么一共可以存储 2^{61} 字节数据）。

客户程序使用的库会缓存 tablet 的位置信息。如果客户程序没有缓存某个 tablet 的地址信息，或者发现它缓存的地址信息不正确，客户程序就在树状的存储结构中递归地查询 tablet 的位置信息。如果客户端缓存是空的，那么寻址算法需要通过三次网络来回通信寻址，其中包括一次 Chubby 读操作。如果客户端缓存的地址信息过期了，那么寻址算法可能需要最多 6 次网络来回通信才能更新数据，因为只有在缓存中没有查到数据时才能发现数据过期（假设 METADATA 的 tablet 没有被频繁地移动）。尽管 tablet 的地址信息是存放在内存里的，对它的操作不必访问 GFS 文件系统，但是，通常会通过预取 tablet 地址来进一步减少访问的开销：每次需要从 METADATA 表中读取一个 tablet 的 metadata 的时候，它都会多读取几个 tablet 的 metadata。

在 METADATA 表中还存储了次级信息，包括每个 tablet 的事件日志（例如，一个服务器什么时候开始为该 tablet 提供服务），这些信息有助于排查错误和性能分析。

2. tablet 分配

在任何一个时刻，一个 tablet 只能分配给一个 tablet 服务器。master 服务器记录了当前有哪些是活跃的 tablet 服务器，哪些 tablet 被分配给了哪些 tablet 服务器，哪些 tablet 还没有被分配。当一个 tablet 还没有被分配，并且刚好有一个 tablet 服务器有足够的空闲空间装载该 tablet 时，master 服务器会给这个 tablet 服务器发送一个装载请求，将该 tablet 分配给这个服务器。

Bigtable 使用 Chubby 来跟踪记录 tablet 服务器的状态。当一个 tablet 服务器启动时，它在 Chubby 的一个指定目录下建立一个有唯一名字的文件，并且获取该文件的独占锁。master 服务器实时监控这个目录（服务器目录），因此 master 服务器能够知道有新的 tablet 服务器加入了。如果 tablet 服务器丢失了 Chubby 上的独占锁——比如由于网络断开导致 tablet 服务器和 Chubby 的会话丢失，它就停止对 tablet 提供服务（Chubby 提供了一种高效的机制，利用这种机制，tablet 服务器能够在不增加网络负担的情况下知道它是否还持有锁）。只要文件还存在，tablet 服务器就会试图重新获得对该文件的独占锁；如果文件不存在了，tablet 服务器就不能再提供服务了，它会自行退出。当 tablet 服务器终止时（比如，集群的管理系统将运行该 tablet 服务器的主机从集群中移除），它会尝试释放它持有的文件锁，这样一来，master 服务器就能尽快把 tablet 分



配到其他 tablet 服务器上。

master 服务器负责检查 tablet 服务器是否已经不再为它的 tablet 提供服务了，如果是，那么要尽快重新分配这些 tablet。master 服务器通过轮询 tablet 服务器文件锁的状态来检测 tablet 服务器何时不再为 tablet 提供服务。如果一个 tablet 服务器报告它丢失了文件锁，或者 master 服务器最近几次尝试和它通信都没有得到响应，那么 master 服务器就会尝试获取该 tablet 服务器文件的独占锁；如果 master 服务器成功获取了独占锁，就说明 Chubby 是正常运行的，而 tablet 服务器要么是宕机了，要么是不能和 Chubby 通信了，因此，master 服务器删除该 tablet 服务器在 Chubby 上的服务器文件，以确保它不再给 tablet 提供服务。一旦 tablet 服务器在 Chubby 上的服务器文件被删除了，master 服务器就把之前分配给它的所有 tablet 放入未分配的 tablet 集合中。为了确保在 master 服务器和 Chubby 之间网络出现故障的时候，Bigtable 集群仍然可以使用，master 服务器在它的 Chubby 会话过期后主动退出。但是不管怎样，如同我们前面所描述的，master 服务器的故障不会改变现有 tablet 在 tablet 服务器上的分配状态。

在集群管理系统启动一个 master 服务器之后，master 服务器首先要了解当前 tablet 的分配情况，之后才能够修改分配。master 服务器在启动的时候执行以下步骤：

- (1) master 服务器从 Chubby 获取唯一的 master 锁，用来阻止创建其他 master 服务器实例。
- (2) master 服务器扫描 Chubby 的服务器文件锁存储目录，获取当前正在运行的服务器列表。
- (3) master 服务器和所有正在运行的 tablet 服务器通信，获取每个 tablet 服务器上 tablet 的分配信息。

(4) master 服务器扫描 METADATA 表获取所有的 tablet 集合。在扫描的过程中，当 master 服务器发现了一个还没有分配的 tablet 时，master 服务器就将这个 tablet 加入未分配的 tablet 集合中，以等待合适的时机进行分配。

可能会遇到一种复杂的情况：在 METADATA 表的 tablet 还没有被分配之前是不能够扫描它的。因此，在开始第 4 步的扫描之前，如果在第 3 步的扫描过程中发现 root tablet 还没有被分配，master 服务器就把 root tablet 加入未分配的 tablet 集合中。这个附加操作确保了 root tablet 会被分配。由于 root tablet 包括了所有 METADATA 的 tablet 的名字，因此 master 服务器在扫描完 root tablet 后，就得到了所有 METADATA 表的 tablet 名字了。

现有的 tablet 集合只有在以下事件发生时才会发生改变：建立了一个新表或删除了一个旧表、两个 tablet 被合并了、一个 tablet 被分割成两个小的 tablet。master 服务器可以跟踪记录所有这些事件，因为除了最后一个事件外的两个事件都是由它启动的。tablet 分割事件需要特殊处理，因为它是由 tablet 服务器启动的。在分割操作完成之后，tablet 服务器通过在 METADATA 表中记录新的 tablet 信息来提交这个操作；当分割操作提交之后，tablet 服务器会通知 master 服务器。如果分割操作已提交的信息没有通知到 master 服务器（可能两个服务器中有一个宕机了），master 服务器在要求 tablet 服务器装载已经被分割的子表的时候会发现一个新的 tablet。通过对



比 METADATA 表中的 tablet 信息，tablet 服务器会发现 master 服务器要求其装载的 tablet 并不完整，因此，tablet 服务器会重新向 master 服务器发送通知信息。

3. tablet 服务

如图 5-3 所示，tablet 的持久化状态信息被保存在 GFS 上，更新操作被提交到 redo 日志中。在这些更新操作中，最近提交的那些被存放在一个排序的缓存中，我们称这个缓存为 memtable；较早的更新被存放在一系列 SSTable 中。为了恢复一个 tablet，tablet 服务器首先从 METADATA 表中读取它的 metadata。tablet 的 metadata 包含了组成这个 tablet 的 SSTable 列表，以及一系列的 redo point。这些 redo point 指向可能包含该 tablet 数据的已提交的日志记录。tablet 服务器把 SSTable 的索引读进内存，然后通过重复 redo point 之后提交的更新来重建 memtable。

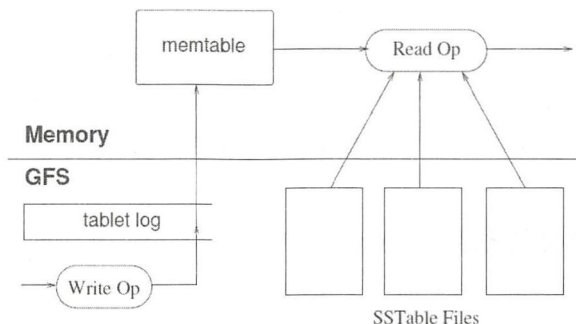


图 5-3 tablet 的工作流程

当对 tablet 服务器进行写操作时，tablet 服务器首先要检查这个操作格式是否正确、操作发起者是否有执行这个操作的权限。权限验证的方法是通过从一个 Chubby 文件里读取出来的具有写权限的操作者列表来进行验证（这个文件几乎一定会被存放在 Chubby 客户缓存里）。成功的修改操作会被记录在提交日志里。可以采用 group commit 的提交方式来提高包含大量小修改操作的应用程序的吞吐量。在一个写操作提交后，写的内容被插入到 memtable 里面。

当对 tablet 服务器进行读操作时，tablet 服务器会进行类似的完整性和权限检查。一个有效的读操作在一个由一系列 SSTable 和 memtable 合并的视图里执行。由于 SSTable 和 memtable 是按字典排序的数据结构，因此可以高效生成合并视图。

当进行 tablet 的合并和分割时，正在执行的读写操作能够继续进行。

4. Compaction（合并）

随着写操作的执行，memtable 的大小不断增加。当 memtable 的大小到达一个限值的时候，这个 memtable 就会被冻结，然后创建一个新的 memtable；被冻结住的 memtable 会被转换成 SSTable，然后写入 GFS。我们称这种 compaction 行为为 minor compaction。minor compaction



有两个目的：减少 tablet 服务器所需要使用的内存，以及在服务器灾难恢复过程中减少必须从提交日志里读取的数据量。在 Compaction 过程中，正在进行的读写操作仍能继续。

每一次 minor compaction 都会创建一个新的 SSTable。如果 minor compaction 过程不停滞地持续进行下去，读操作可能需要合并来自多个 SSTable 的更新；否则，我们通过定期在后台执行 merging compaction 过程合并文件，来限制这类文件的数量。merging compaction 过程读取一些 SSTable 和 memtable 的内容，合并成一个新的 SSTable。只要 merging compaction 过程完成了，输入的这些 SSTable 和 memtable 就可以删除了。

合并所有的 SSTable 并生成一个新的 SSTable 的 merging compaction 过程叫作 major compaction。由非 major compaction 产生的 SSTable 可能含有特殊的删除条目，这些删除条目在旧的、但是依然有效的 SSTable 中可能还会存在。而 major compaction 过程生成的 SSTable 不包含已经删除的信息或数据。Bigtable 循环扫描它所有的 tablet，并且定期对它们执行 major compaction。major compaction 机制允许 Bigtable 回收已经删除的数据占有的资源，并且确保 Bigtable 能及时清除已经删除的数据，这对存放敏感数据的服务是非常重要的。

5.2.3 Bigtable 的性能优化

为了使 Bigtable 达到用户要求的高性能、高可用性和高可靠性，我们还需要做很多优化工作才行。本节将讲解 Bigtable 的优化工作。

1. Locality Group（本地组）

客户程序可以将多个 column family 组合成一个 locality group。对 tablet 中的每个 locality group 都会生成一个单独的 SSTable。将通常不会一起访问的列族分割成不同的局部性群组可以提高读取操作的效率。例如，Webtable 中的网页的 metadata（比如 language 和 checksums）可以在一个 locality group 中，网页的内容可以在另一个 locality group 中。当一个应用程序要读取网页的 metadata 的时候，它没有必要去读取所有的页面内容。

此外，可以以 locality group 为单位设定一些有用的调试参数。比如，可以把一个 locality group 设定为全部存储在内存中。tablet 服务器依照懒加载的策略将设定为放入内存的 locality group 的 SSTable 装载进内存。在加载完成之后，在访问属于该 locality group 的 column family 的时候就不必读取硬盘了。这个特性对于需要频繁访问的小块数据特别有用。在 Bigtable 内部，我们利用这个特性来提高在 METADATA 表中具有位置相关性的 column family 的访问速度。

2. 压缩

客户程序可以控制一个 locality group 的 SSTable 是否需要压缩，以及当需要压缩时，以什



么格式来压缩。每个 SSTable 的块（块的大小由 locality group 的优化参数指定）都使用用户指定的压缩格式来压缩。虽然分块压缩浪费了少量空间，但是，我们在只读取 SSTable 的一小部分数据的时候就不必解压整个文件了。很多客户程序采用了“两遍”的、可定制的压缩方式。第一遍采用 Long Common Strings 的方式，这种方式在一个很大的扫描窗口里对常见的长字符串进行压缩，可以参见 Jon Bentley 和 Douglas McIlroy 发表的论文 *Data Compression Using Long Common Strings*（在线地址为 <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.8470&rep=rep1&type=pdf>）。第二遍是采用快速压缩算法，即在一个 16 KB 的小扫描窗口中寻找重复数据。两个压缩的算法都很快，在现在的机器上，压缩的速率达到 100~200 MB/s，解压的速率达到 400~1000 MB/s。

在选择压缩算法的时候重点考虑的是速度而不是压缩的空间，但是这种“两遍”的压缩方式在空间压缩率上的表现也令人惊叹。比如，在 Webtable 的例子中，我们使用这种压缩方式来存储网页内容。在一次测试中，我们压缩了一个局部的群组中存储了大量的网页。针对实验的目的，我们没有存储每个文档所有版本的数据，仅仅存储了一个版本的数据。该模式的空间压缩比达到了 10:1。这比传统的 Gzip 在压缩 HTML 页面时 3:1 或者 4:1 的空间压缩比好得多；“两遍”的压缩模式如此高效的原因是由于 Webtable 的行的存放方式：从同一个主机获取的页面都存在临近的地方。利用这个特性，Long Common Strings 的方式可以从来自同一个主机的页面里找到大量的重复内容。不仅仅是 Webtable，其他的很多应用程序也通过选择合适的行名来将相似的数据聚簇在一起，以获取较高的压缩率。当我们在 Bigtable 中存储同一份数据的多个版本的时候，压缩效率会更高。

3. 缓存

为了提高读操作的性能，tablet 服务器使用二级缓存的策略。Scan Cache 是第一级缓存，主要缓存 tablet 服务器通过 SSTable 接口获取的 key-value 对；Block Cache 是二级缓存，缓存的是从 GFS 读取的 SSTable 的数据块。对于应用程序来说，假设经常要重复读取相同数据，则 Scan Cache 非常有效；假设经常要读取刚刚读过的数据附近的数据（即数据之间的存储位置比较接近），则 Block Cache 更有用，例如，顺序读或者在一个热点的 row 的 locality group 中随机读取不同的 column。

4. Bloom 过滤器

一个读操作必须读取构成 tablet 状态的所有 SSTable 的数据。如果这些 SSTable 不在内存中，就需要多次访问硬盘。我们通过允许客户程序对特定 locality group 的 SSTable 指定 Bloom 过滤器，来减少硬盘访问的次数。我们可以使用 Bloom 过滤器查询一个 SSTable 是否包含了特定 row 和 column 的数据。对于某些特定应用程序，我们只付出了少量的、用于存储 Bloom 过滤器的内存的代价，就换来了读操作显著减少的磁盘访问的次数。使用 Bloom 过滤器也隐式地达到了



当应用程序访问不存在的 row 或 column 时，大多数时候不需要访问硬盘的目的。

有关 Bloom 过滤器的详细内容，可以参阅 https://en.wikipedia.org/wiki/Bloom_filter。

5. Commit 日志的实现

如果我们把对每个 tablet 的操作的 commit 日志都存在一个单独的的文件的话，就会产生大量的文件，并且这些文件会并行地写入 GFS。根据 GFS 服务器底层文件系统实现的方案，要把这些文件写入不同的磁盘日志文件时，会有大量的磁盘 seek 操作。另外，由于 group commit 方式在提交过程中，操作的数目一般比较少，因此，对每个 tablet 设置单独的日志文件也会给批量提交本应具有优化效果带来很大的负面影响。为了避免这些问题，我们设置每个 tablet 服务器一个 commit 日志文件，把修改操作的日志以追加方式写入同一个日志文件中，因此在一个实际的日志文件中混合了对多个 tablet 修改的日志记录。

使用单个日志显著提高了普通操作的性能，但是将恢复的工作复杂化了。当一个 tablet 服务器宕机时，它加载的 tablet 将会被移到很多其他 tablet 服务器上：每个 tablet 服务器都装载很少的几个原来的服务器的 tablet。当恢复一个 tablet 的状态的时候，新的 tablet 服务器要从原来的 tablet 服务器写的日志中提取修改操作的信息，并重新执行。然而，这些 tablet 修改操作的日志记录都混合在同一个日志文件中的。一种方法是从新的 tablet 服务器读取完整的 commit 日志文件，然后只重复执行它需要恢复的 tablet 的相关修改操作。使用这种方法，假如有 100 台 tablet 服务器，每台都加载了失效的 tablet 服务器上的一个 tablet，那么，这个日志文件要被读取 100 次（每个服务器读取一次）。

为了避免多次读取日志文件，我们首先把日志按照关键字进行排序：

(table, row name, log sequence number)

在排序之后，对同一个 tablet 的修改操作的日志记录就连续存放在了一起，因此，我们只要进行一次磁盘 seek 操作之后顺序读取就可以了。为了并行排序，我们先将日志分割成 64 MB 的段，之后在不同的 tablet 服务器上对段进行并行排序。这个排序工作由 master 服务器来协同处理，并且在一个 tablet 服务器表明自己需要从 commit 日志文件恢复 tablet 时开始执行。

在向 GFS 中写 commit 日志的时候可能会引起系统性能波动，原因是多种多样的，比如，写操作正在进行的时候，一个 GFS 服务器宕机了，或者连接三个 GFS 副本所在的服务器的网络拥塞/过载了。为了确保在 GFS 负载高峰时修改操作还能顺利进行，每个 tablet 服务器实际上有两个日志写入线程，每个线程都写自己的日志文件，并且在任何时刻，只有一个线程是工作的。如果一个线程在写入的时候效率很低，tablet 服务器就切换到另外一个线程，修改操作的日志记录就被写入这个线程对应的日志文件中。每个日志记录都有一个序列号，因此，在恢复的时候，tablet 服务器能够检测出并忽略掉那些由于线程切换而导致的重复记录。

6. 加快 tablet 恢复

当 master 服务器将一个 tablet 从一个 tablet 服务器移到另外一个 tablet 服务器时，源 tablet 服务器会对这个 tablet 做一次 minor compaction。这个 minor compaction 操作减少了 tablet 服务器的日志文件中没有合并的记录，从而减少了恢复的时间。minor compaction 完成之后，该服务器就停止为该 tablet 提供服务。在卸载 tablet 之前，源 tablet 服务器还会再做一次 minor compaction（通常会很快），以消除前面在一次压缩过程中又产生的未合并的记录。在第二次 minor compaction 完成以后，tablet 就可以被装载到新的 tablet 服务器上了，并且不需要从日志中进行恢复。

7. 利用不变性

在使用 Bigtable 时，除了 SSTable 缓存，其他部分产生的 SSTable 都是不变的，我们可以利用这一点对系统进行简化。例如，当从 SSTable 读取数据的时候，不必对文件系统访问操作进行同步。这样一来，就可以非常高效地实现对行的并行操作。memtable 是唯一一个能被读和写操作同时访问的可变数据结构。为了减少读操作时的竞争，我们对内存表采用 COW（Copy-on-write）机制，这样就允许读写操作并行执行。

因为 SSTable 是不变的，因此，我们可以把永久删除被标记为“删除”的数据问题，转换成对废弃的 SSTable 进行垃圾收集的问题。每个 tablet 的 SSTable 都在 METADATA 表中注册，master 服务器采用“标记—删除”的垃圾回收方式删除 SSTable 集合中废弃的 SSTable，METADATA 表则保存了 root SSTable 的集合。

要在一个小的 Bigtable 集群删除过时的文件，master 会定期扫描相应的目录树，来收集其集群中的所有现有文件的名称，然后扫描 METADATA 表，以获得集群中所有的活跃的文件，最后删除存在于前者而不在后者的任何文件（METADATA 表的 row 包含了 tablet 所有文件名称的存储数据）。此过程不会扩展，因此在一个更大的 Bigtable 集群中，master 将这个过程分成小块，这样只需要读取在目录树和 METADATA 表中的部分内容，就能检测和删除其中的死文件。

最后，SSTable 的不变性使得分割 tablet 的操作非常快捷。我们不必为每个分割出来的 tablet 建立新的 SSTable 集合，而是共享原来的 tablet 的 SSTable 集合。

5.3 Apache HBase

Apache HBase 是一个分布式的、面向列的开源数据库。正如上一节所提到的，该技术来源于 Google 的 Bigtable。就像 Bigtable 利用了 GFS 所提供的分布式数据存储一样，Apache HBase（后简称 HBase）在 Hadoop 之上提供了类似于 Bigtable 的能力。HBase 是 Apache 的 Hadoop

项目的子项目。HBase 不同于一般的关系数据库，它是一个适合于非结构化数据存储的数据库，它是基于列的。

5.3.1 Apache HBase 的基本概念

HBase 中的数据被存储在表中，具有行和列。这和关系数据库（RDBMS 中）的术语是重叠的，但在概念上它们不是一类。相反，应该将 HBase 的表当作一个多维的 map 结构，这样更容易让人理解。

1. 术语

- **Table (表)**：HBase table 由多个 row 组成。
- **Row (行)**：每一个 row 代表一个数据对象，每一个 row 都是以一个 row key（行键）和一个或者多个 column 组成的。row key 是每个数据对象的唯一标识的，按字母顺序排序，即 row 也是按照这个顺序进行存储的。所以，row key 的设计相当重要，一个重要的原则是，相关的 row 要存储在接近的位置。比如网站的域名，row key 就是域名，在设计时要将域名反转（例如，org.apache.www、org.apache.mail、org.apache.jira），这样的话，Apache 相关的域名在 table 中存储的位置就会非常接近。
- **Column (列)**：column 由 column family 和 column qualifier 组成，由冒号 (:) 进行间隔，比如 family:qualifier。
- **Column Family (列族)**：在 HBase 中，column family 是一些 column 的集合。一个 column family 的所有 column 成员有着相同的前缀。比如，courses:history 和 courses:math 都是 courses 的成员。冒号 (:) 是 column family 的分隔符，用来区分前缀和列名。column 前缀必须是可打印的字符，剩下的部分列名可以是任意字节数组。column family 必须在 table 建立的时候声明。column 随时可以新建。在物理上，每个 column family 成员在文件系统上都是存储在一起的。因为存储优化都是针对 column family 级别的，这就意味着，一个 column family 的所有成员都是用相同的方式访问的。
- **Column Qualifier (列限定符)**：column family 中的数据通过 column qualifier 来进行映射。column qualifier 也没有特定的数据类型，以二进制字节来存储。比如某个 column family “content”，其 column qualifier 可以被设置为 “content:html” 和 “content:pdf”。虽然 column family 在 table 创建时就固定了，但 column qualifier 是可变的，在不同的 row 之间可能有很大不同。
- **Cell (单元格)**：cell 是 row、column family 和 column qualifier 的组合，包含了一个值

和一个 timestamp，用于标识值的版本。

- **Timestamp (时间戳)**：每个值都会有一个 timestamp，作为该值特定版本的标识符。在默认情况下，timestamp 代表了数据被写入 RegionServer 的时间，但也可以在把数据放到 cell 时指定不同的 timestamp。

2. map

HBase/Bigtable 的核心数据结构就是 map。不同的编程语言针对 map 有不同的术语，比如 associative array (PHP)、associative array (Python)、Hash (Ruby) 或 Object (JavaScript)。

简单来说，map 就是 key-value 对集合。下面是用 JSON 格式来表达 map 的例子：

```
{
  "zzzzzz" : "woot",
  "xyz" : "hello",
  "aaaab" : "world",
  "1" : "x",
  "aaaaa" : "y"
}
```

3. 分布式

毫无疑问，HBase/Bigtable 都是建立在分布式系统上的，HBase 基于 Hadoop Distributed File System (HDFS) 或者 Amazon's Simple Storage Service (S3)，而 Bigtable 使用 Google File System (GFS)。它们要解决数据同步问题。这里不讨论如何做到数据同步。HBase/Bigtable 可以部署在成千上万的机器上来分散访问压力。

4. 排序

和一般的 map 实现有所区别，HBase/Bigtable 中的 map 是按字母顺序严格排序的。也就是说，row key 在“aaaaa”的旁边，应该是“aaaab”，与“zzzzz”离得较远。

还是以上面的 JSON 为例，一个排好序的例子如下：

```
{
  "1" : "x",
  "aaaaa" : "y",
  "aaaab" : "world",
  "xyz" : "hello",
  "zzzzzz" : "woot"
}
```

在一个大数据量的系统里面，排序很重要，特别是 row key 的设置策略决定了查询的性能。

比如网站的域名，row key 就是域名，在设计时要将域名反转（例如，org.apache.www、org.apache.mail、org.apache.jira）。

5. 多维

多维 map，即 map 里面嵌套 map。例如：

```
{
  "1" : {
    "A" : "x",
    "B" : "z"
  },
  "aaaaa" : {
    "A" : "y",
    "B" : "w"
  },
  "aaaab" : {
    "A" : "world",
    "B" : "ocean"
  },
  "xyz" : {
    "A" : "hello",
    "B" : "there"
  },
  "zzzzz" : {
    "A" : "woot",
    "B" : "1337"
  }
}
```

6. 时间版本

在查询中不指定时间，返回的将是最近一个时间的版本。如果给出 timestamp，则返回的将是早于这个时间的数值。例如：查询 row/column 是 “aaaaa” / “A:foo” 的，将返回 y；查询 row/column/timestamp 是 “aaaaa” / “A:foo” /10 的，将返回 m；查询 row/column/timestamp 是 “aaaaa” / “A:foo” /2 的，将返回 null。

```
{
  // ...
  "aaaaa" : {
```



```
"A" : {
  "foo" : {
    15 : "y",
    4 : "m"
  },
  "bar" : {
    15 : "d",
  }
},
"B" : {
  "" : {
    6 : "w"
    3 : "o"
    1 : "w"
  }
},
// ...
}
```

7. 概念视图

表 5-1 是一个名为 weetable 的 table，包含了两个 row (com.cnn.www 和 com.example.www) 和三个 column family (contents、anchor 和 people)。在第一个 row (com.cnn.www) 中，anchor 包含了两个 column (anchor:cssnsi.com 和 anchor:my.look.ca)，contents 包含一个 column (contents:html)。在这个例子里面，row key 是 com.cnn.www 的 row 包含了 5 个版本，而 row key 是 com.example.www 的 row 包含 1 个版本。column qualifier 为 contents:html 包含给定网站的完整 HTML。column family 是 anchor 的每个 qualifier 包含网站的链接。column family 是 people 关联的网站的人物资料。

表 5-1 名为 weetable 的 table

Row Key	Timestamp	ColumnFamily contents	ColumnFamily anchor	ColumnFamily people
"com.cnn.www"	t9		anchor:cnnsi.com = "CNN"	
"com.cnn.www"	t8		anchor:my.look.ca = "CNN.com"	
"com.cnn.www"	t6	contents:html = "..."		

续表

Row Key	Timestamp	ColumnFamily contents	ColumnFamily anchor	ColumnFamily people
"com.cnn.www"	t5	contents:html = "..."		
"com.cnn.www"	t3	contents:html = "..."		

在这个表中显示为空的 cell 不占用空间，这使得 HBase 变得“稀疏”。除了用表格方式来展现数据视图，也可以使用多维 map：

```
{
  "com.cnn.www": {
    contents: {
      t6: contents:html: "<html>...",
      t5: contents:html: "<html>...",
      t3: contents:html: "<html>...",
    }
    anchor: {
      t9: anchor:cnnsi.com = "CNN",
      t8: anchor:my.look.ca = "CNN.com"
    }
    people: {}
  }
  "com.example.www": {
    contents: {
      t5: contents:html: "<html>..."
    }
    anchor: {}
    people: {
      t5: people:author: "John Doe"
    }
  }
}
```

8. 物理视图

尽管在概念视图里，table 可以被看作一个稀疏的 row 的集合，但在物理上，它是按照 column family 存储的。新的 column qualifier (column_family:column_qualifier) 可以随时添加进已有的 column family 中。

表 5-2 是一个 ColumnFamily anchor。

表 5-2 ColumnFamily anchor

Row Key	Timestamp	Column Family anchor
"com.cnn.www"	t9	anchor:cnnsi.com = "CNN"
"com.cnn.www"	t8	anchor:my.look.ca = "CNN.com"

表 5-3 是一个 ColumnFamily contents。

表 5-3 ColumnFamily contents

Row Key	Timestamp	ColumnFamily contents:
"com.cnn.www"	t6	contents:html = "..."
"com.cnn.www"	t5	contents:html = "..."
"com.cnn.www"	t3	contents:html = "..."

值得注意的是，在上面的概念视图中，空白 cell 在物理上是不存储的，因为根本没有必要存储。因此若一个请求为要获取 t8 时间的 contents:html，则它的结果是空。相似地，若请求为获取 t9 时间的 anchor:my.look.ca，则结果也是空。但是，如果不指明 timestamp，则将会返回最新时间的 column。例如，如果请求为获取行键为“com.cnn.www”，且没有指明 timestamp，则返回的结果是 t6 下的 contents:html、t9 下的 anchor:cnnsi.com 和 t8 下的 anchor:my.look.ca。

9. 数据模型操作

四个主要的数据模型操作是 Get、Put、Scan 和 Delete，通过 Table 实例进行操作。有关 Table 的 API 可以参见 <http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Table.html>。

Get

Get 返回特定 row 的属性，通过 Table.get 执行。有关 Get 的 API 可以参见 <http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Get.html>。

Put

Put 向 table 增加新 row（如果 key 是新的）或更新 row（如果 key 已经存在），通过 Table.put（writeBuffer）或 Table.batch（非 writeBuffer）执行。有关 Put 的 API 可以参见 <http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Put.html>。

Scan

Scan 允许多个 row 特定属性迭代。

下面是一个在 Table 表实例上的 Scan 示例。假设 table 有几行 row key 为“row1”“row2”和“row3”，还有一些 row key 值为“abc1”“abc2”和“abc3”。下面的示例展示 Scan 实例

如何返回以“row”打头的 row。

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes();
...

Table table = ...           // 实例化一个 Table 实例

Scan scan = new Scan();
scan.addColumn(CF, ATTR);
scan.setRowPrefixFilter(Bytes.toBytes("row"));
ResultScanner rs = table.getScanner(scan);
try {
    for (Result r = rs.next(); r != null; r = rs.next()) {
        // ...
    }
} finally {
    rs.close(); // 最终关闭 ResultScanner
}
```

注意，通常最简单的方法指定 scan 停止点采用 InclusiveStopFilter 类，其 API 可以参见 <http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/InclusiveStopFilter.html>。

Delete

Delete 用于从 table 中删除 row，通过 Table.delete 来执行操作。有关 Delete 的 API 可以参见 <http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Delete.html>。

HBase 没有修改数据的合适方法。所以 Delete 通过创建名为 tombstones 的新标志进行处理。这些 tombstones 和死去的值会在 major compaction 时清除掉。

10. Namespace（命名空间）

在 HBase 中，namespace 指对一组 table 的逻辑分组，类似 RDBMS 中的 database，方便对 table 在业务上进行划分。这种抽象奠定了 multi-tenancy（多租户）相关功能的基础。

- Quota Management（配额管理）——限制 namespace 的资源（比如，region、table）的使用。有关该功能的详细描述可参见 <https://issues.apache.org/jira/browse/HBASE-8410>；
- Namespace Security Administration（命名空间安全管理）——提供租户另一个层次的安全管理。有关该功能的详细描述可参见 <https://issues.apache.org/jira/browse/HBASE-9206>；
- Region 服务器组——命名 namespace/table 可以固定到某个 RegionServer 的子集，从而

保证了隔离。有关该功能的详细描述可参见 <https://issues.apache.org/jira/browse/HBASE-6721>。

namespace 管理

namespace 可以被创建、删除、修改。table 和 namespace 的隶属关系在创建 table 时决定，通过以下格式来指定：

```
<table namespace>:<table qualifier>
```

下面是示例：

```
#Create a namespace
create_namespace 'my_ns'
```

```
#create my_table in my_ns namespace
create 'my_ns:my_table', 'fam'
```

```
#drop namespace
drop_namespace 'my_ns'
```

```
#alter namespace
alter_namespace 'my_ns', {METHOD=> 'set', 'PROPERTY_NAME'=> 'PROPERTY_VALUE'}
```

预定义的 namespace

HBase 系统默认预定义了 namespace：

- hbase——系统 namespace，用于包括 HBase 内置表；
- default——用户建表时未指定 namespace 的 table 都默认使用该 namespace。

下面是示例：

```
#namespace=foo and table qualifier=bar
create 'foo:bar', 'fam'
```

```
#namespace=default and table qualifier=bar
create 'bar', 'fam'
```

5.3.2 Apache HBase 的架构

关于 HBase 的架构，MapR 公司的架构师 Carol McDonald 在其博客 *An In-Depth Look at the HBase Architecture*（网址为 <https://www.mapr.com/blog/in-depth-look-hbase-architecture>）中进行

了非常深入的讲解。在物理上，HBase 遵循的是 master-slave 模式，由三种类型的服务器组成。其中，Region 服务器提供数据的读取和写入。访问数据时，客户端直接与 RegionServer 交互；Region 的分配、DDL 操作（创建、删除 table）由 HBase Master（HMaster）线程处理；ZooKeeper 作为 HDFS 的一部分，用于维护现场的集群状态。

Hadoop DataNode 存储 Region Server 在管理过程中的数据。所有 HBase 的数据存储在 HDFS 文件中。Region Server 会配备 HDFS DataNode，这使得 Region Server 服务的数据可以存储于本地（接近需要的地方）。HBase 输入的数据是在本地的，但是如果一个 Region 被移除，则必须等待 compaction 以后才能恢复到本地。

NameNode 会维护所有组成该文件的物理数据块的元数据信息。

HBase 的架构组成如图 5-4 所示。

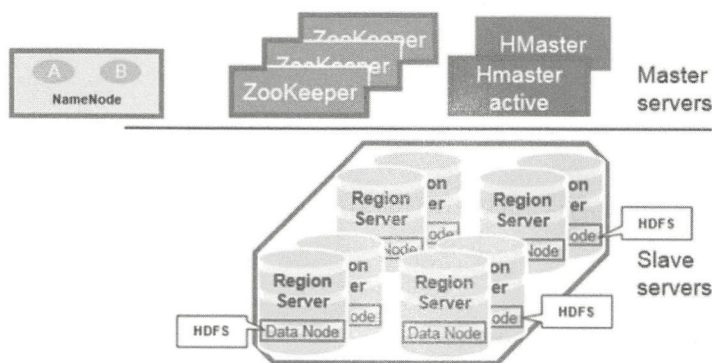


图 5-4 HBase 的架构组成

1. Region

HBase 使用 row key 将 table 水平切割成多个 Region，每个 Region 都记录它的 start key 和 end key，第一个 Region 的 start key 和最后一个 Region 的 end key 为空。由于 row key 是排序的，因而 Client 可以通过 HMaster 快速定位每个 row key 在哪个 Region 中。Region 由 HMaster 分配到集群节点相应的 RegionServer 中，然后由 RegionServer 负责 Region 的启动和管理，以及和 Client 的通信，负责数据的读写。每个 RegionServer 都可以同时管理 1000 个左右的 Region。

Region 在架构中的位置如图 5-5 所示。

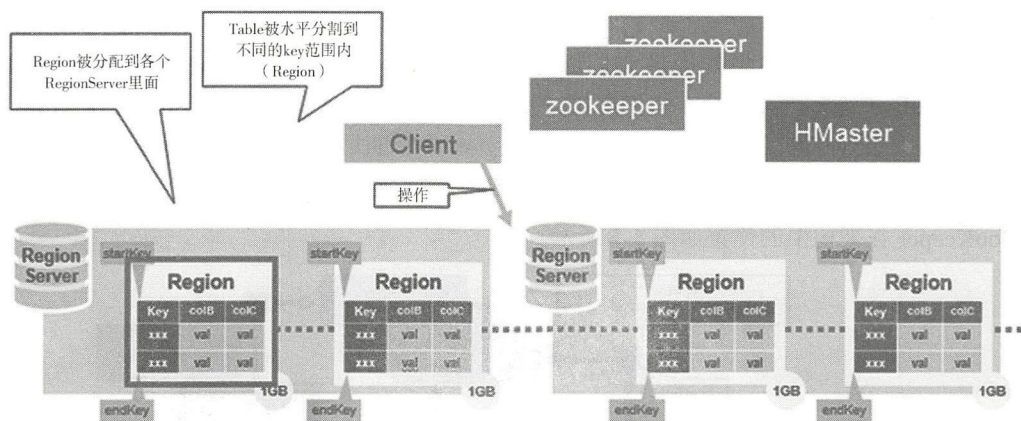


图 5-5 Region 在架构中的位置

2. HMaster

Region 的分配、DDL 操作（创建、删除 table）由 HMaster 线程处理。它主要有两方面的职责：

- 协调 Region 服务器；
- 启动时 Region 的分配，以及进行负载均衡和修复时 Region 的重新分配；
- 监控集群中所有 RegionServer 实例的状态(通过 heartbeat 和监听 ZooKeeper 中的状态)。
- 管理——提供创建、删除、修改 table 的接口。

HMaster 在架构中的位置如图 5-6 所示。

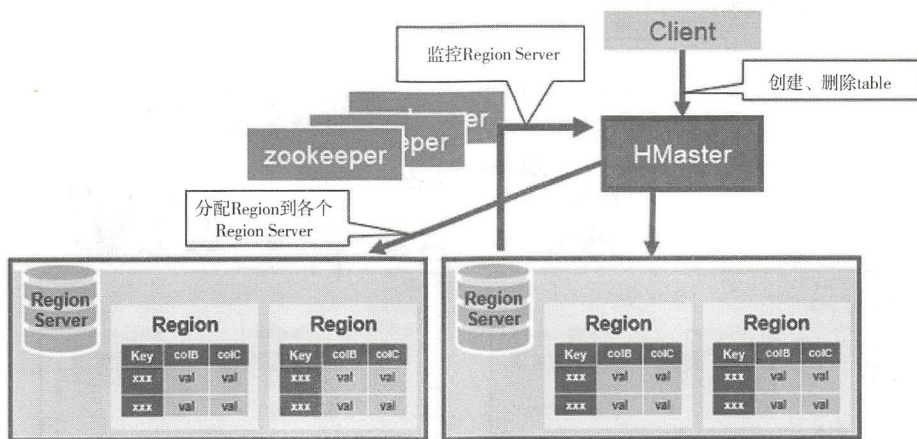


图 5-6 HMaster 在架构中的位置

3. ZooKeeper

ZooKeeper 为 HBase 集群提供协调服务，它管理着 HMaster 和 RegionServer 的状态（available、alive 等），并且会在它们宕机时发通知给 HMaster，从而 HMaster 实现 HMaster 之间的 failover，或对宕机的 RegionServer 中的 Region 集合的修复。ZooKeeper 集群本身使用一致性协议来保证每个节点状态的一致，需要注意的是，ZooKeeper 至少要有 3 到 5 台机器。

ZooKeeper 在架构中的位置如图 5-7 所示。

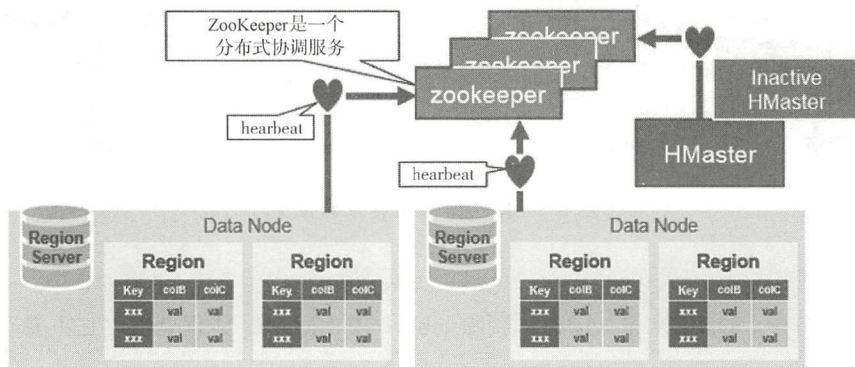


图 5-7 ZooKeeper 在架构中的位置

4. 各个组件的协作

ZooKeeper 协调集群所有节点的共享信息，唯一一个 active 的 HMaster 和 Region Server 通过 session 连接到 ZooKeeper。ZooKeeper 使用 heartbeat 机制维持这些 ephemeral node（瞬息的节点）的存活状态，如果某个节点失效，则 HMaster 会收到通知，并做相应的处理。

架构中各个组件的协作如图 5-8 所示。

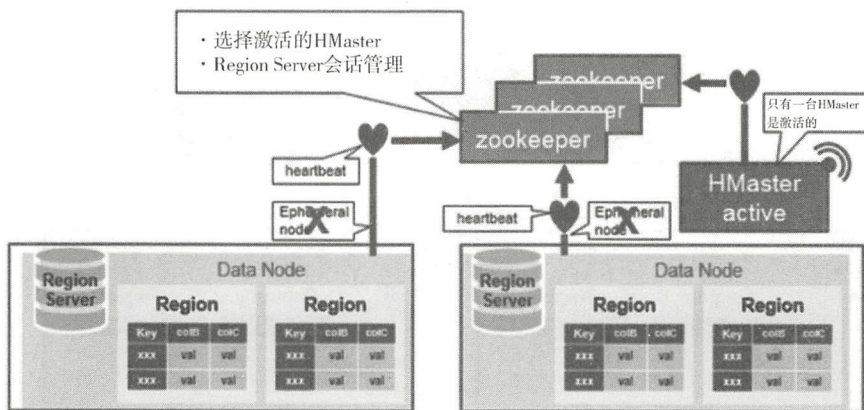


图 5-8 架构中各个组件的协作

每个 Region Server 都会创建一个 ephemeral node, HMaster 通过监听这些节点来监控 Region Server 是否可用, 以及服务器是否故障。多个 HMaster 会竞争来创建一个 ephemeral node, 由 ZooKeeper 来判断哪个 HMaster 是第一个, 并确保该 HMaster 是唯一一个 active 的 HMaster, Zookeeper 与该 HMaster 通过 heartbeat 进行维持。其后加进来的 HMaster 则监听该 active 的 HMaster 的状态, 如果当前 active 的 HMaster 或 Region Server 发送 heartbeat 失败, 那么 session 会超时, 相应的 ephemeral node 就会被删除, 因而其他 HMaster 得到通知, 接替成为 active 的 HMaster。

5. HBase 的第一次读写

有个特殊的 HBase Catalog table 被称为 META table, 它保存集群中的 Region 的位置信息。而 ZooKeeper 保存 META table 的位置信息。

当客户端第一次对 HBase 进行读写时, 将发生:

- (1) 客户端获得存储在 ZooKeeper META table 中的 Region Server;
- (2) 客户端将查询.META.服务器, 根据 row key 来获取对应的 Region Server, 并缓存 META table 的位置信息;
- (3) 从查询到的 Region Server 中读取 row。

对于下一次的读取, 客户端使用缓存检索 META 的位置和读取 row key 即可。随着时间的推移, 客户端缓存的位置信息越来越多, 以至于它不需要再查询 META table, 除非有未命中(可能因为 Region 发生变动了), 它才会重新查询和更新缓存。

HBase 第一次读写的流程如图 5-9 所示。

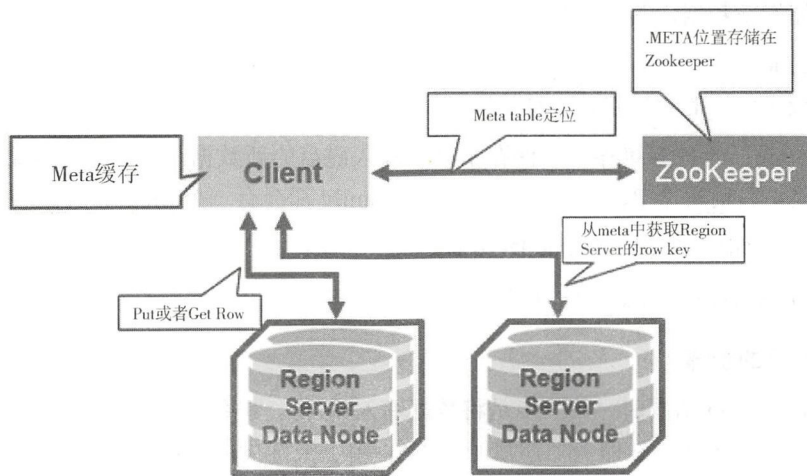


图 5-9 HBase 第一次读写的流程

Meta Table 用于保存所有 Region 的列表，它就像一个 B 树，其结构为：

- Key——Region 的 start key、Region id；
- Value——RegionServer。

Meta Table 的结构如图 5-10 所示。

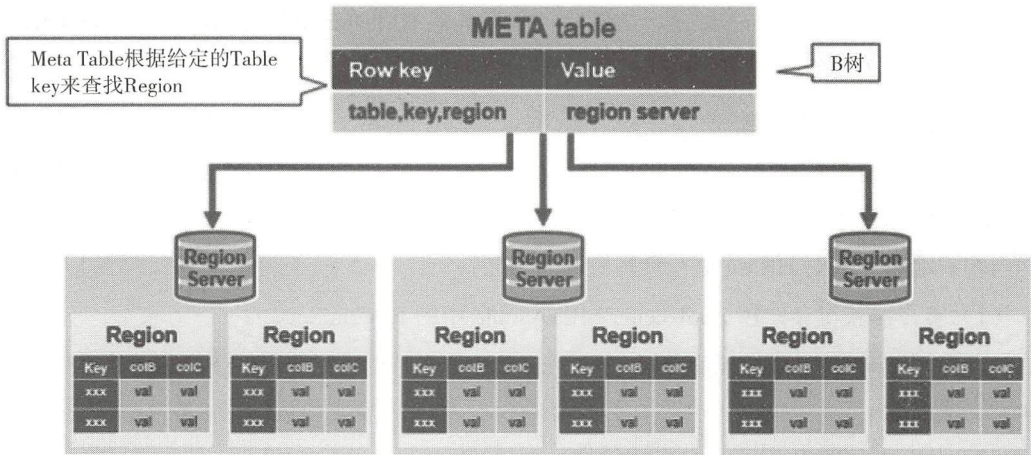


图 5-10 Meta Table 的结构

6. Region Server 组件

Region Server 运行在 HDFS 数据节点上，由下面的组件构成。

- WAL（Write Ahead Log）：WAL 是分布式文件系统上的文件，用于存储尚未保存到持久存储的新数据，其作用是在出现故障的情况下实现系统恢复。
- BlockCache：一种读缓存。将数据预读取到内存中，以提升读的性能。当缓存满了以后，最近最少使用的数据将首先被清除。
- MemStore：一种写缓存。它保存了尚未写入磁盘的新数据。它在被写入磁盘之前会进行排序。目前每个 Region 的每个 column family 都会有一个 MemStore。
- HFile：存储排序成 key-value 的 row。

Region Server 的构造如图 5-11 所示。

7. HBase 写的步骤

当客户端发出 Put 请求时，第一步是将数据写入 WAL。其中，编辑文件被附加到存储在磁盘上的 WAL 文件的末尾。在服务器崩溃时，WAL 用于未持久化的数据的恢复。

客户端发出 Put 请求，如图 5-12 所示。

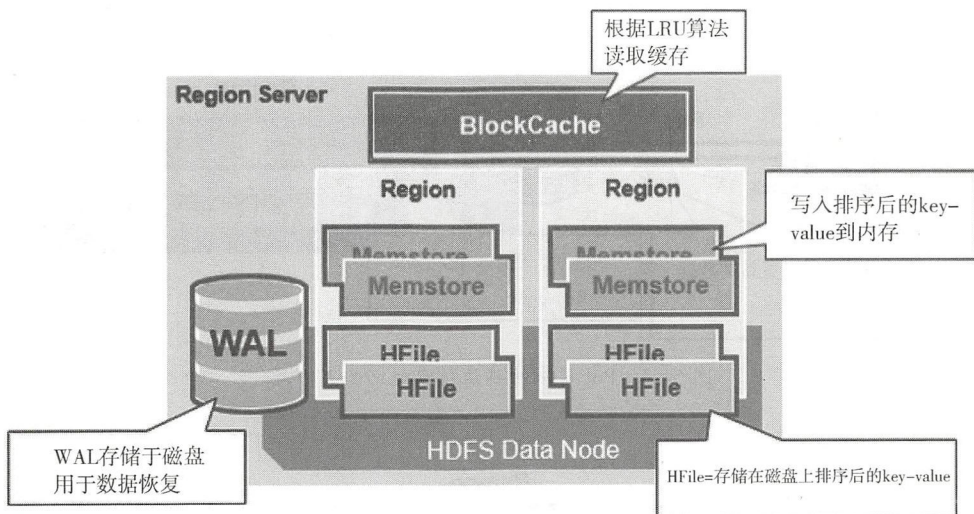


图 5-11 Region Server 的构造

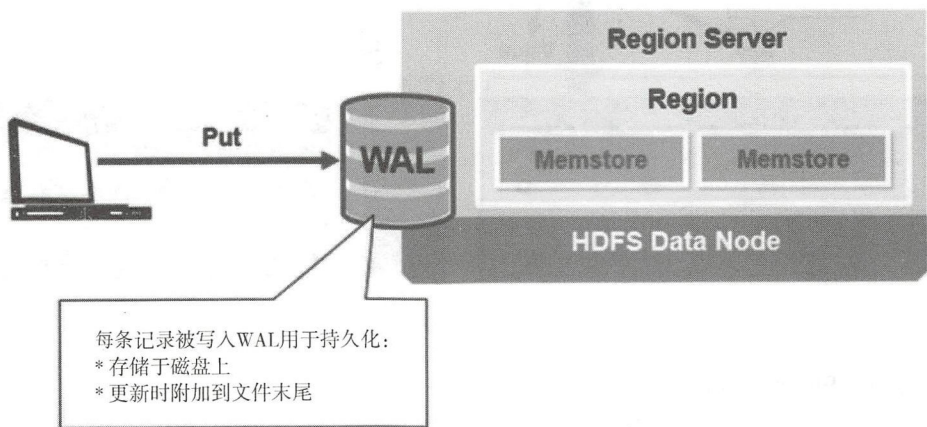


图 5-12 客户端发出 Put 请求

一旦数据被写入 WAL，它就被放置在 MemStore 中。然后，Put 请求的确认信息将返回给客户端，如图 5-13 所示。

8. MemStore

MemStore 存储更新内容到内存中并以 key value 进行分类，同样，这些内容将被存储在一个 HFile 中。每个 column family 都有一个 MemStore。这些更新是按 column family 来分类的。

MemStore 示意图如图 5-14 所示。

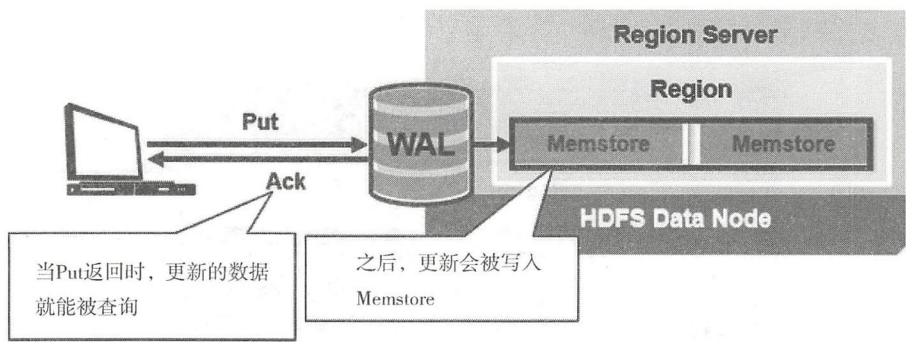


图 5-13 Put 请求的确认信息返回给客户端

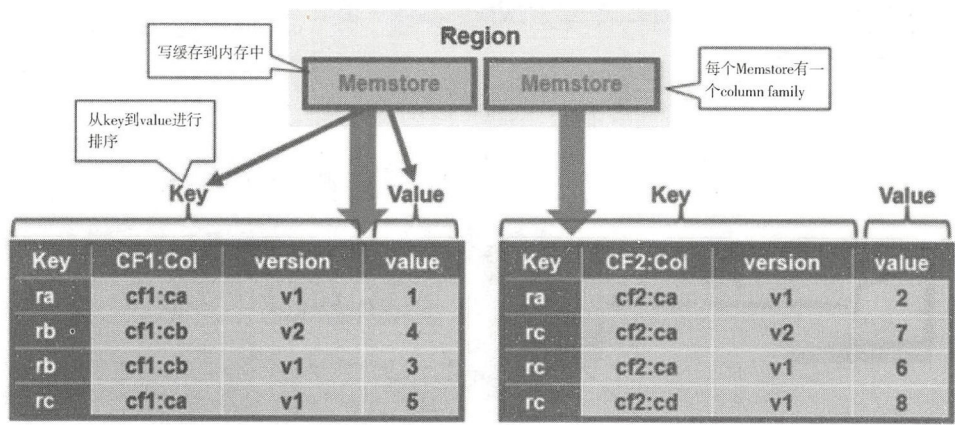


图 5-14 MemStore 示意图

9. Region Flush（冲刷）

当 MemStore 积累了足够的数据后，整个有序集合会被写入 HDFS 的一个新 HFile 中。每个 column family 都会有多个 HFile，其中包含实际的 cell 或 key value 实例。随着时间的推移，MemStore 中的这些文件会被 flush 到磁盘文件。

请注意，column family 是有数量限制的，其原因是，每个 column family 都有一个 MemStore，当 MemStore 满时，就 flush。它也可以节省最后写入的序列号，以便系统知道当前在执行的内容。

最高的序列号作为一个 meta field（元字段）被存储在 HFile 中，以反映持续在哪里结束，在哪里继续。当 Region 启动时，会先读取序列号，最高的序列号被用作新编辑文件的序列号。

Region Flush 示意图如图 5-15 所示。

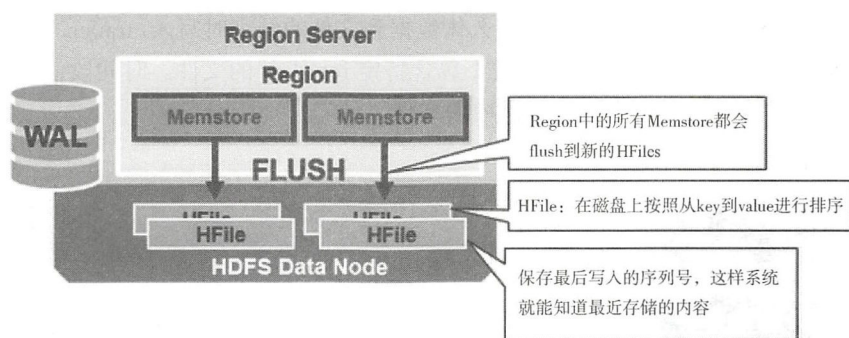


图 5-15 Region Flush 示意图

10. HFile

HBase 的数据以 key value 的形式顺序存储在 HFile 中, 在 MemStore 的 flush 过程中生成 HFile, 由于 MemStore 中存储的 cell 遵循相同的排列顺序, 因而 flush 过程是顺序写, 所以写数据到磁盘的性能很高, 因为不需要不停地移动 disk drive head (磁盘驱动器磁头)。

HFile 示意图如图 5-16 所示。

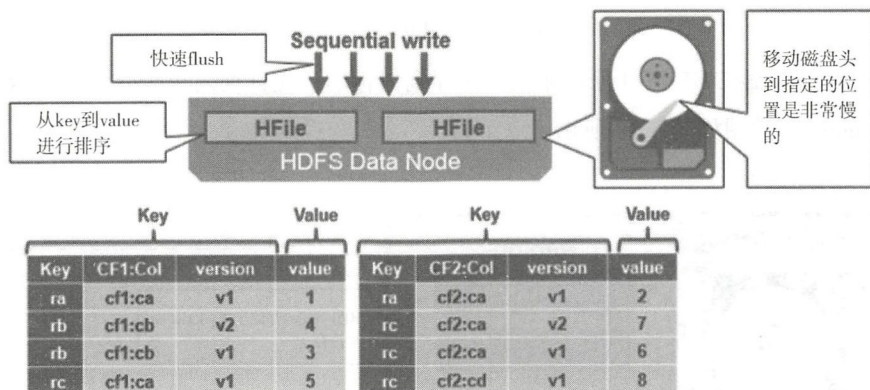


图 5-16 HFile 示意图

一个 HFile 包含多层 index (索引) 并允许 HBase 来查找数据, 从而避免了读取整个文件。多层索引像 B+树:

- key value 以递增的顺序存储;
- 索引所指向的数据被存放在 64KB 的“块”里面;
- 每个块都有自己的 leaf-index (叶子索引);
- 每个块的最后一个 key 被用于放置 intermediate index (中间索引);
- root index (根索引) 指向 intermediate index。

trailer 指向 meta block(元块), 并在持久化数据到文件的结束时写入。trailer 可以包含 bloom filter 和时间段信息。bloom filter 帮助跳过不包含特定 row key 的文件。时间段信息用于在查找特定时间段的文件时, 跳过不在该时间段内的文件。

HFile 结构如图 5-17 所示。

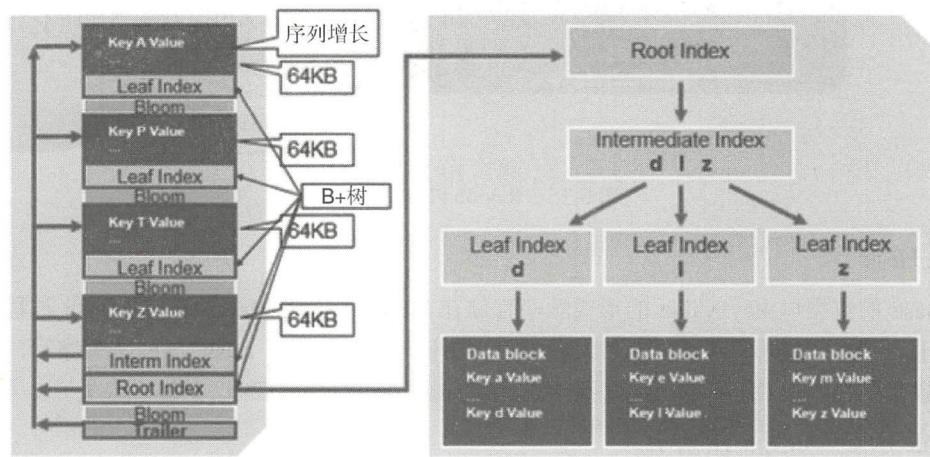


图 5-17 HFile 结构

index（索引）是在 HFile 打开时加载并保存到内存的，允许使用单个磁盘进行查找。
HFile index 示意图如图 5-18 所示。

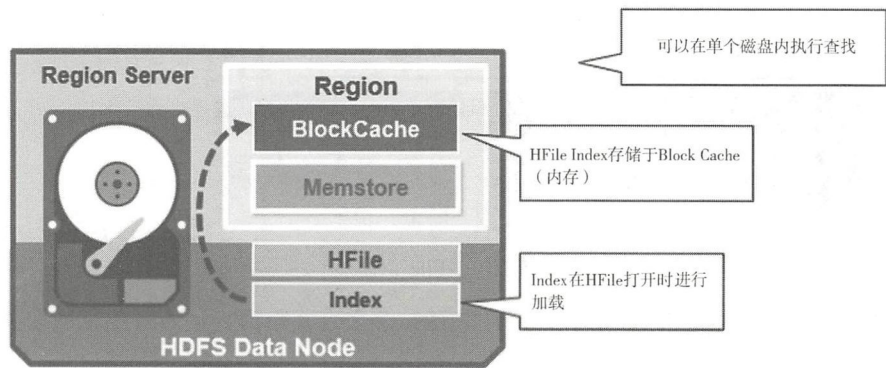


图 5-18 HFile index 示意图

11. HBase 读合并

对应同一个 row 的 cell 并不保证在一起: row cell 持久化在 HFile 中; 最近更新的 cell 存在于 MemStore 中; 最近读的 cell 存在于 Block cache 中。既然相同的 cell 可能存储在三个地方, 在读取的时候需要扫描这三个地方, 然后将结果合并即可 (Read Merge), 在 HBase 中扫描的

顺序依次是：Block cache、MemStore、HFile。

(1) 扫描 Block cache，读取缓存。最近读取的 key value 会缓存到这里，最近最少使用的会在内存满时被清除掉。

(2) 扫描 MemStore。里面存储的是最近写入的数据。

(3) 若在 MemStore 和 Block Cache 中扫描不到 row cell，则会使用 Block Cache 索引和 bloom filter 来加载 HFile 文件到内存中，里面就包含了需要的目标文件 row cell。

HBase 读的流程如图 5-19 所示。

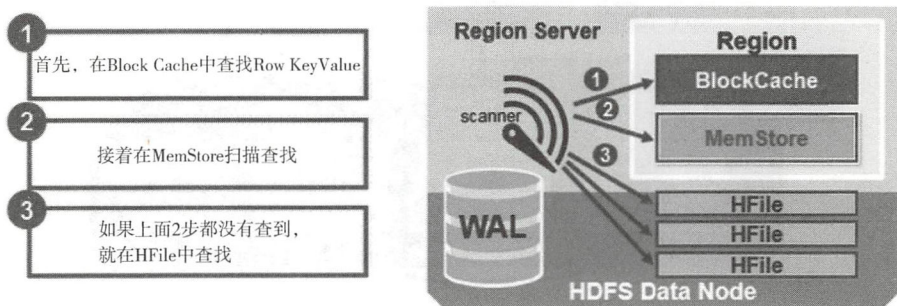


图 5-19 HBase 读的流程

MemStore 每次 flush 都会创建新的 HFile，而过多的 HFile 会引起读的性能问题。HBase 采用 Compaction 机制来解决这个问题。

12. Minor Compaction

HBase 会自动选取一些小的 HFile，将它们重写合并成一个更大的 HFile，这个过程下被称为 Minor Compaction。这个过程中不会处理标记为 deleted 或者 expired 的 cell。一次 Minor Compaction 的结果是更少并且更大的 HFile。

Minor Compaction 示意图如图 5-20 所示。

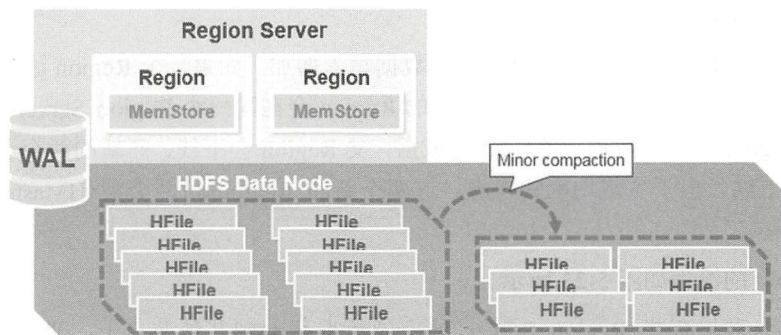


图 5-20 Minor Compaction 示意图

13. Major Compaction

Major Compaction 是指将所有的 HFile 合并成一个 HFile，在这个过程中，标记为 deleted 或者 expired 的 cell 会被丢弃，这样就改进了读的性能。然而由于它会引起很多的磁盘 I/O 操作和网络负担而引起性能问题（写入放大），因而它一般会被安排在周末、凌晨等集群比较闲的时间。

Major Compaction 示意图如图 5-21 所示。

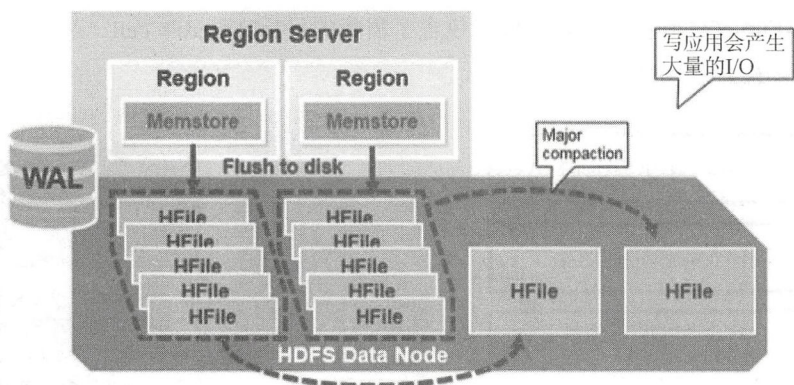


图 5-21 Major Compaction 示意图

14. Region

让我们对 Region 做一个快速的回顾：

- 一个 table 可以水平地被分成一个或多个 Region。一个 Region 包含 start key 和 end key 有序区间内、连续的 row；
- 每个 Region 默认是 1 GB 的大小；
- table 的 Region 以 Region Server 的形式服务于客户端；
- 一个 Region Server 可以服务于大约 1000 个 Region（可能属于同一个 table 或不同的 table）。

Region 的组成如图 5-22 所示。

最初，一个 table 只有一个 Region，随着数据写入增加，如果一个 Region 达到一定的大小，就需要 Split（分割）成两个 Region。两个新的 Region 会在同一个 Region Server 中创建，它们各自包含父 Region 一半的数据，当 Split 完成后，父 Region 会下线，而新的两个子 Region 会向 HMaster 注册上线。出于负载均衡的考虑，这两个新的 Region 可能会被 HMaster 分配到其他 Region Server 中。

Region Split 的流程如图 5-23 所示。

负载均衡会引起有些 Region Server 处理的数据移动到其他节点上，直到下一次 Major Compaction 才将数据从远端的节点移动到本地节点。

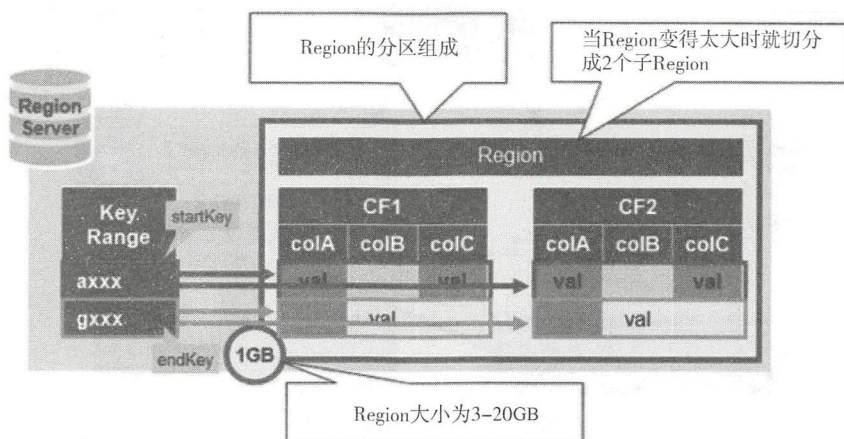


图 5-22 Region 的组成

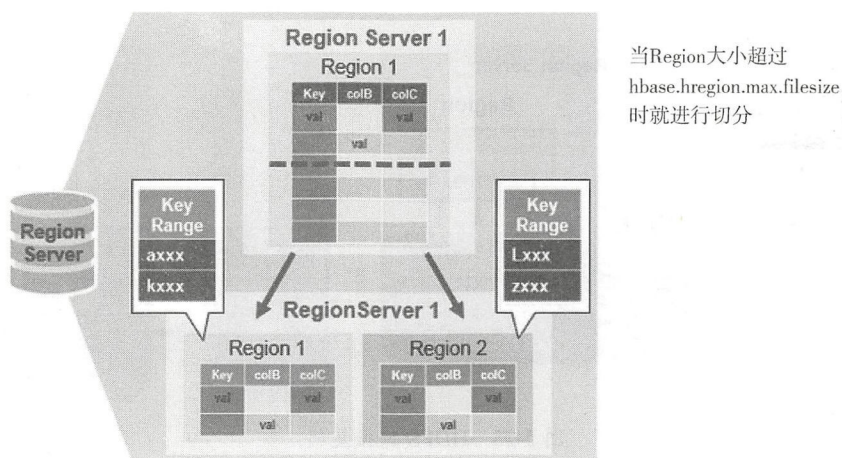


图 5-23 Region Split 的流程

Region 负载均衡如图 5-24 所示。

15. HDFS 数据复制

所有写入和读取都是操作主节点。HDFS 会复制 WAL 和 HFile 块。HFile 块的复制是自动发生的。HBase 依赖于 HDFS 作为其存储的文件时提供的数据安全性。当数据被写入 HDFS 时，一个副本在本地被写入，然后将其复制到 secondary node（二级节点），而第三个副本被写入 tertiary node（三级节点）。

HDFS 数据复制如图 5-25 所示。

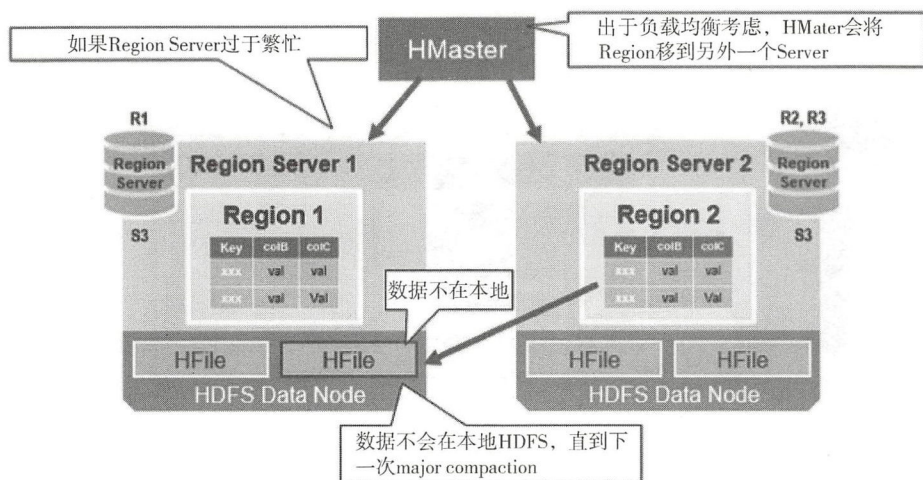


图 5-24 Region 负载均衡

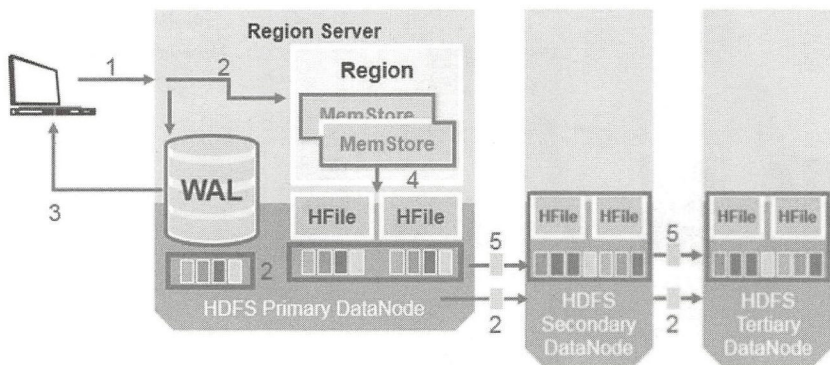


图 5-25 HDFS 数据复制

16. Region 故障恢复

当一台 Region Server 故障时, 由于它不再发送 heartbeat 给 ZooKeeper 而被监测到, 此时 ZooKeeper 会通知 HMaster, HMaster 会检测到哪台 Region Server 发生了故障, 它将故障的 Region Server 中的 Region 重新分配给其他的 Region Server, 同时 HMaster 会把故障的 Region Server 相关的 WAL 拆分并分配给相应的 Region Server (将拆分出的 WAL 文件写入对应的目的 Region Server 的 WAL 目录中, 并写入对应的 DataNode 中), 从而这些 Region Server 可以 replay (重播) 分到的 WAL 来重建 MemStore。

Region 故障恢复如图 5-26 所示。

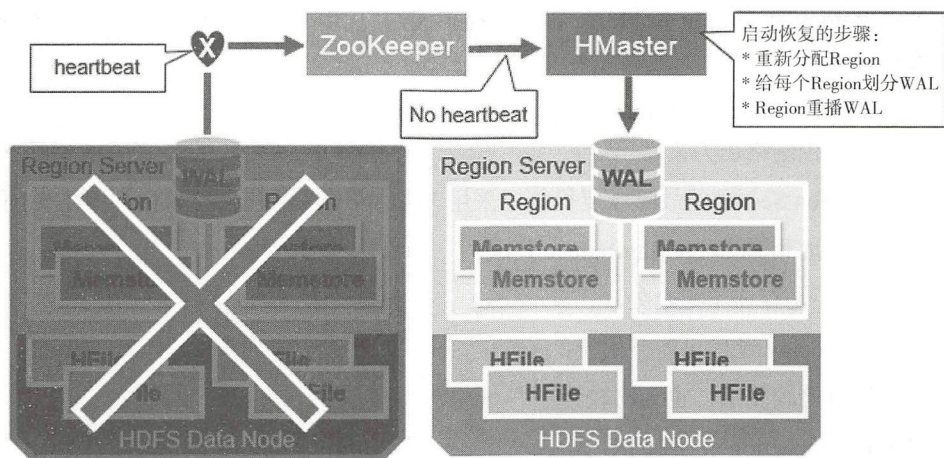


图 5-26 Region 故障恢复

17. 数据恢复

WAL 文件包含编辑列表，一个编辑代表一个 Put 或 Delete。编辑按时间顺序写入，所以在存储时会附加到磁盘上的 WAL 文件的末尾。

如果出现故障时数据仍然在内存中，而且没有保存到一个 HFile 中，此时 WAL 文件就会 replay 并通过读 WAL，添加和排序所包含的编辑到当前的 MemStore。最后，MemStore 被 flush 到 HFile 中。

数据恢复如图 5-27 所示。

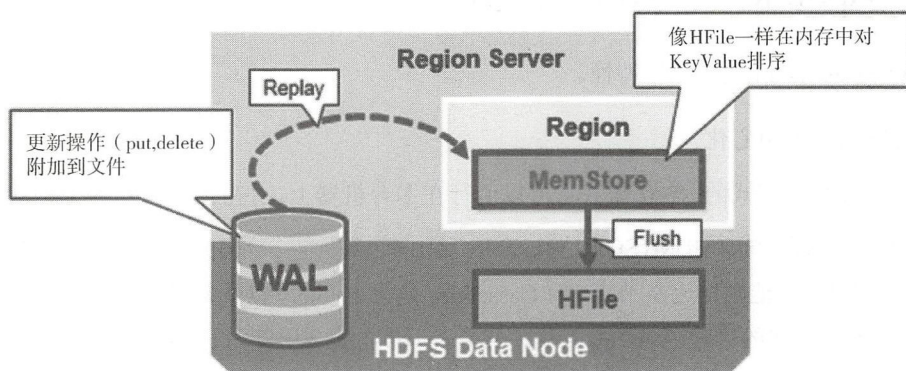


图 5-27 数据恢复

从上面的 HBase 架构的设计可以总结出该设计实现上的优点：

- HBase 采用强一致性模型，在一个写返回后，保证所有的读操作都能读到相同的数据；



- 通过 Region 动态 Split 和 Merge 实现自动扩展, 并使用 HDFS 提供的多个数据备份功能, 实现高可用性;
- 采用 Region Server 和 DataNode 运行在相同的服务器上实现数据的本地化, 提升读写性能, 并减少网络压力;
- 内建 Region Server 的故障恢复机制。采用 WAL 来 replay 还未持久化到 HDFS 的数据;
- 可以无缝地和 Hadoop/MapReduce 集成。

当然该架构还是存在不足之处:

- WAL 的 replay 过程可能会很慢;
- 故障恢复比较复杂, 也会比较慢;
- Major Compaction 会引起 I/O 风暴。

5.4 Apache Cassandra

Apache Cassandra 是一个开源的、分布式、去中心化、弹性可扩展、高可用性、容错、一致性可调、面向行的数据库, 它基于 Amazon Dynamo 的分布式设计和 Google Bigtable 的数据模型。它最初由 Facebook 创建, 用于储存收件箱等简单格式的数据, 于 2008 年开源。此后, 由于 Cassandra 良好的可扩展性, 被 Digg、Twitter 等知名 Web 2.0 网站所采纳, 成为一种流行的分布式结构化数据存储方案。

5.4.1 Apache Cassandra 简介

Apache Cassandra 具有如下特性。

1. 分布式与去中心化

Cassandra 是分布式的, 这意味着它可以运行在多台机器上, 并呈现给用户一个一致的整体。

对于很多存储系统 (如 MySQL、Bigtable、HBase 等), 一旦开始扩展它, 就需要把某些节点设为主节点, 其他则作为从节点。但 Cassandra 是去中心的, 也就是说每个节点都是一样的, 没有节点会承担特殊的管理任务。与主从模式 (master-slave) 相反, Cassandra 的协议是 P2P 的, 并使用 gossip 来维护存活或死亡节点的列表。

去中心化这一事实意味着 Cassandra 不会存在单点失效。Cassandra 集群中的所有节点的功能都完全一样, 所以不存在一个特殊的主机作为主节点来承担协调任务。有时这被叫作 “server symmetry (服务器对称)”。





综上所述，因为 Cassandra 是分布式、去中心化的，它不会有单点失效的问题，所以支持高可用性。

2. 弹性可扩展

可扩展性是指系统架构可以让系统提供更多的服务而不降低使用性能的特性。最简单的达到可扩展性的手段，就是给现有的机器增加硬件的容量、内存来进行垂直扩展。而水平扩展则需要增加更多机器，每台机器提供全部或部分数据，这样所有主机都不必负担全部业务请求，但软件自己需要有内部机制来保证集群中节点间的数据同步。

弹性可扩展是指水平扩展的特性，即集群可以在不间断的情况下，方便扩展或缩减服务的规模。这样，用户就不需要重新启动进程，不必修改应用的查询，也无须自己手工重新均衡数据分布。在 Cassandra 里，只要加入新的计算机，Cassandra 就会自动地发现它并让它开始工作。

3. 高可用与容错

系统的可用性是由满足请求的能力来量度的。但计算机可能会有各种各样的故障，从硬件器件故障到网络中断都有可能。所以它们一般都有硬件冗余，并在发生故障事件的情况下会自动响应并进行热切换。从软件的层次来说，设置多个数据中心，就是“软件冗余”，它能保障在灾难发生时，故障中断的功能在剩余系统上恢复。

Cassandra 就是高可用的。用户可以在不中断系统的情况下替换故障节点，还可以把数据分布到多个数据中心里，从而提供更好的本地访问性能，并且在某一数据中心发生火灾、洪水等不可抗灾难的时候防止系统彻底瘫痪。

4. Tuneable Consistency（可调一致性）

一致性的基本含义是读操作一定会返回最新写入的结果。扩展数据存储系统就意味着我们不得不在数据一致性、节点可用性和分区容错性之间做某些折中（即 CAP 理论）。Cassandra 常被称为是“最终一致性”的，简单地说，Cassandra 牺牲了一点一致性来换取完全的可用性。但是 Cassandra 实际更应该表述为“可调一致性”，它允许用户方便地选定需要的一致性水平与可用性水平，在二者之间找到平衡点。

客户端可以控制在更新到达多少个副本之前，必须阻塞系统。通过设置 replication factor（副本因子）来调节与之相对的一致性级别。通过 replication factor，用户可以决定准备牺牲多少性能来换取一致性。replication factor 是用户要求更新在集群中传播到的节点数（注意，更新包括所有增加、删除和更新操作）。

客户端每次操作还必须设置一个 consistency level（一致性级别）参数，这个参数决定了多少个副本写入成功才可以认定写操作是成功的，或者读取过程中读到多少个副本正确就可以认





定是读成功的。这里，Cassandra 把决定一致性程度的权利留给了用户自己。

所以，如果需要的话，用户就可以设定 consistency level 和 replication factor 相等，从而达到一个较高的一致性水平，不过这样必须付出同步阻塞操作的代价，只有所有节点都被更新完成才能成功返回一次更新。而实际上，Cassandra 一般都不会这么用，原因显而易见（这样就丧失了可用性目标，影响性能，而且这不是用户选择 Cassandra 的初衷）。如果一个客户端设置 consistency level 低于 replication factor 的话，则即使有节点宕机，仍然可以写成功。

5. Row-Oriented（面向行）

Cassandra 不是真正意义上的“Column-Oriented（面向列）”的数据库，而是“Row-Oriented（面向行）”的，它的数据结构不是关系型的，而是一个多维稀疏哈希表。“稀疏”意味着任何一行都可能会有一列或者几列，但每行都不一定（像关系模型那样）和其他行有一样的列。每行都有一个唯一的键值，用于进行数据访问。所以，更确切地说，应该把 Cassandra 看作一个有索引的、Row-Oriented 的存储系统。

Cassandra 的数据存储结构基本可以被看作一个多维哈希表。这意味着不必事先精确地决定具体数据结构或是记录应该包含哪些具体字段。这特别适合处于草创阶段，还在不断增加或修改服务特性的应用，而且特别适合应用在敏捷开发项目中，不必进行长达数月的预先分析。对于使用 Cassandra 的应用，如果业务发生了变化了，只需要在运行中增加或删除某些字段就行了，不会造成服务中断。

当然，这不是说不需要考虑数据。相反，Cassandra 需要我们换个角度看数据。在 RDBMS 里，首先得设计一个完整的数据模型，然后考虑查询方式，而在 Cassandra 里可以首先思考如何查询数据，然后提供这些数据。

6. Flexible Schema（灵活的模式）

早期版本的 Cassandra 忠实于 Bigtable 的设计，采用的是“schema-free（无模式）”的数据模型，新的 column 可以被动态定义。schema-free 模式的数据库在处理大数据时有非常强的扩展和高性能的优势。但这种模式的主要缺点是在确定数据的含义和数据的格式时存在难点，而这限制了执行复杂查询的能力。

Cassandra Query Language（CQL）正是为了解决上面提到的问题而产生的。CQL 提供了一种通过类似于 Structured Query Language（SQL）的语法来定义的模式。起初，CQL 是基于 Apache Thrift 项目的 schema-free 接口来提供额外的 Cassandra 接口的。在这个过渡阶段，模式是可选的，可以通过 CQL 来定义，也可以通过 Thrift API 在添加新的 column 时动态扩展。

自 Cassandra 3.0 以来，将不推荐采用通过基于 Thrift API 来实现动态创建 column 的方式。Cassandra 的底层存储已重新实现，并与 CQL 更紧密地结合起来。Cassandra 并不限制动态扩展





模式，但它的工作方式已经显著不同了。CQL 的集合，如 list、set，特别是 map 具有在无结构化的格式里面添加内容的功能，从而能扩展现有的模式。CQL 还可以改变列的类型，以支持存储 JSON 格式的文本。

因此，Cassandra 支持“灵活的模式”。

7. 高性能

Cassandra 在设计之初就特别考虑了要充分利用多处理器和多核计算机的性能，并考虑在分布于多个数据中心的这类大量服务器上运行。它可以一致而且无缝地扩展到数百台机器。Cassandra 已经显示出了高负载下的良好表现，在一个非常普通的工作站上，Cassandra 也可以提供非常高的写吞吐量。如果增加更多的服务器，则还可以继续保持 Cassandra 所有的特性而无须牺牲性能。

5.4.2 Apache Cassandra 的应用场景

尽管 Cassandra 设计精巧、功能出色，但也并非能胜任所有工作。所以，下面介绍 Cassandra 最擅长的领域。

1. 大规模部署

Cassandra 的很多精巧设计都专注于高可用、可调一致性、P2P 协议、无缝扩展等，这些都是 Cassandra 的“卖点”。这些特性在单节点工作时都是没有意义的，更无法展现它的全部能力。所以，我们需要做一些评估，考虑期望的流量、吞吐需求及 SLA 等。如果要求数据库可以很好地应付流量，提供不错的性能，可能选关系型数据库会更好。简单地说，这是因为 RDBMS 更易于在单机上运行，对用户来说也更熟悉。

但是，如果认为需要至少几个节点才能支撑业务，则 Cassandra 是不错的选择。如果应用可能需要数十个节点，则 Cassandra 可能就是很很棒的选择了。

2. 写密集、统计和分析型工作

Cassandra 是为优异的写吞吐量而特别优化的。许多早期使用 Cassandra 的产品都用于存储用户状态更新、社交网络、建议/评价以及应用统计等。这些应用大都是写多于读的，而且更新可能随时发生并伴有突发的峰值。事实上，支撑应用负载需要很高的多客户线程并发写性能，这正是 Cassandra 的主要特性。

Cassandra 已经被用于开发了多种不同的应用，包括窗口化的时间序列数据库，用于文档搜索的反向索引，以及分布式任务优先级队列。





3. 地区分布

Cassandra 支持多地分布，可以很容易地配置成将数据分布到多个数据中心的存储方式。如果有一个全球部署的应用，那么让数据贴近用户会获得不错的性能收益，Cassandra 正适合这种应用场合。

4. 变化的应用

如果正在“初创阶段”，业务会不断改进，Cassandra 这种灵活的数据模型可能更适合，这让数据库更快地跟上业务改进的步伐。

5.4.3 Apache Cassandra 的架构和数据模型

Cassandra 是设计来处理跨多个节点的大数据工作负载，且不会产生单点故障问题。Cassandra 通过在集群中的所有节点之间分布均匀的节点，采用 P2P 的方式来定位分布式系统涉及的故障问题。每个节点经常交流有关其自身的状态信息，其他节点利用 P2 gossip 通信协议与集群进行交互。Commit 日志在每个节点上会依次写入活动信息，以确保数据的持久性，然后数据被索引并写入内存结构 memtable 中。memtable 类似于一个回写高速缓存。每次当这个内存结构满时，数据就被写入磁盘中的 SSTable 数据文件中。所有的写操作都会自动分区，并在整个集群中进行复制。Cassandra 定期使用一种被称为 compaction 过程来合并 SSTable，合并过程中会丢弃标记为 tombstone 的数据。为了确保所有的数据在整个集群中保持一致，将使用不同的修复机制。

Cassandra 是一个分区 row 存储数据库，其中 row 被组织成一个必须有主键的 table。Cassandra 的架构允许任何已授权的用户使用 CQL 语言来连接到任何数据中心的任何节点。为了方便使用，CQL 使用类似的 SQL 的语法来操作表数据。开发人员可以通过 cqlsh、DevCenter 或者应用语言的驱动程序来访问 CQL。在通常情况下，集群的每个应用程序都会有一个 keyspace（密钥空间），由许多不同的 table 组成。

客户端读取或写入请求可以被发送到集群中的任何节点。当客户端通过一个请求连接到一个节点时，则该节点充当协调器用于该客户端特定的操作。协调器充当客户端应用程序和拥有被请求数据的节点之间的代理。协调器来决定在哪些节点能收到请求，这往往是基于集群的配置要求。

Cassandra 包含如下核心构件。

- **Node（节点）**：存储数据的地方，是 Cassandra 的基础设施组件。
- **Data center（数据中心）**：相关节点的集合。数据中心可以是物理数据中心或虚拟数据中心。不同的工作负载应使用单独的数据中心，无论物理的还是虚拟的。复制是由数





据中心设置的。使用独立的数据中心可以防止 Cassandra 事务受到其他进程的影响，并保持彼此接近为较低的延迟请求。根据不同的复制因子，可将数据写入到多个数据中心。数据中心必须永远跨越物理位置。

- **Cluster (集群)** : 一个集群包含一个或多个数据中心，可以跨越物理位置。
- **Commit log (Commit 日志)** : 所有数据首先被写入 Commit 日志以确保数据的持久性。当其所有的数据已冲刷到 SSTable 时，它可以存档、删除或回收。
- **SSTable**: SSTable (sorted string table, 经排序的字符串表) 是 Cassandra 定期写 memtable 的不变的数据文件。SSTable 仅追加并在磁盘上依次存储并保持 Cassandra table。
- **CQL Table**: 通过 table row 获取的有序的 column 集合。table 由 column 和一个主键组成。

5.4.4 用于配置 Apache Cassandra 的核心组件

1. Gossip

P2P 通信协议用于发现和分享在 Cassandra 集群中的其他节点的位置和状态信息。Gossip 信息也被保存在每个节点本地，当一个节点重新启动时可以被立即使用。

2. Partitioner (分区工具)

Partitioner 用于确定哪些节点先接收第一个数据片段的副本，以及如何分配其他副本到集群的其他节点上。数据的每一 row 由唯一的主键来识别，该主键可以与 partition key (分区键) 相同，但也可能包括了其他 clustering column。一个 partitioner 是一个哈希函数，用于从 row 的主键中派生一个 token。Partitioner 使用 token 值，以确定集群中哪些节点可以接收该 row 的副本。Murmur3Partitioner 是新的 Cassandra 集群默认的分区策略，在大部分情况下也是在新的集群中的正确选择。

必须设置 Partitioner 并为每个节点分配 num_tokens 值。token 的分配数量取决于系统的硬件功能。如果不能使用 virtual nodes (vnodes, 虚拟节点)，则请使用 initial_token 设置代替。

3. Replication factor (副本因子)

Replication factor 是指集群副本的总数。replication factor 为 1 意味着一个节点上每个 row 仅有一个副本；为 2 意味着每个 row 有 2 个副本，且每个副本位于不同节点上。所有副本都同样重要，不区分是否是主副本。用户需要为每个数据中心定义 Replication factor。通常应设置策略大于 1，但不超过集群中的节点总数。





4. Replica placement strategy（副本放置策略）

Cassandra 存储数据的副本到多个节点，以确保可靠性和容错性。复制策略决定哪些节点需要放置到副本上。第一个数据的副本是简单的第一副本，它并不是唯一的。强烈建议在大多数部署情况下使用 `NetworkTopologyStrategy`，因为它在需要时更容易扩展到多个数据中心。

在创建一个 `keyspace`（密钥空间）后，必须定义副本放置策略和副本数。

5. Snitch

Snitch 定义了数据中心的计算机组以及复制策略用于放置副本的机架（拓扑）。

当创建一个集群时，必须配置一个 Snitch。所有 Snitch 使用动态 Snitch 层来监控性能，并选择最佳副本用于读取。它默认是启用的，并推荐在大多数部署情况下使用。在 `cassandra.yaml` 配置文件里面来配置每个节点的动态 Snitch 阈值。

`SimpleSnitch` 默认不能识别的数据中心或机架的信息，可以在公共云的单数据中心部署或单区中使用。建议在生产环境中使用 `GossipingPropertyFileSnitch`，它定义了一个节点的数据中心和机架，并且使用 `gossip` 来传播该信息到其他节点。

6. cassandra.yaml 配置文件

主配置文件用于设置集群的初始化属性，比如 `table` 参数的缓存、调节属性和资源利用率、超时设置、客户端连接、备份、安全等。

在默认情况下，节点被配置用于存储受其管理的且设置在 `cassandra.yaml` 文件目录下的数据。

在集群的生产部署环境，可以修改 `commitlog-directory` 目录到与 `data_file_directories` 不同的磁盘驱动下。

7. 系统 keyspace table 属性

可以通过基本的编程方式来设置每个 `keyspace` 或 `table` 的存储配置属性，或者使用客户端应用，比如 CQL。

5.5 Memcached

Memcached 是一个高性能的分布式内存对象缓存系统，用于动态 Web 应用以减轻数据库负载。它通过在内存中缓存数据和对象来减少读取数据库的次数，从而提高动态、数据库驱动网站的速度。Memcached 是基于一个存储 `key-value` 对的 `hashmap`，其守护进程是用 C 语言编写的，但是客户端可以用任何语言来编写，并通过 `memcached` 协议与守护进程通信。



5.5.1 Memcached 简介

Memcached 作为高速运行的分布式缓存服务器，具有以下特点：

- 协议简单；
- 基于 libevent 的事件处理；
- 内置内存存储方式；
- Memcached 不互相通信的分布式。

Memcached 主要由以下几部分组成。

- 客户端软件：能够获取到可用的 Memcached 服务器的列表；
- 基于客户端的散列算法：它会根据“key”来选择服务器；
- 服务器软件：存储 key-value 到内部哈希表；
- LRU：Least Recently Used（最近最少使用算法）来决定何时要清除掉旧数据（如果内存不足的话），或重复使用内存。

5.5.2 Memcached 的架构

Memcached 拥有非常精简的架构设计理念，它可以按需来更好地利用内存。Memcached 的工作原理如图 5-28 所示。

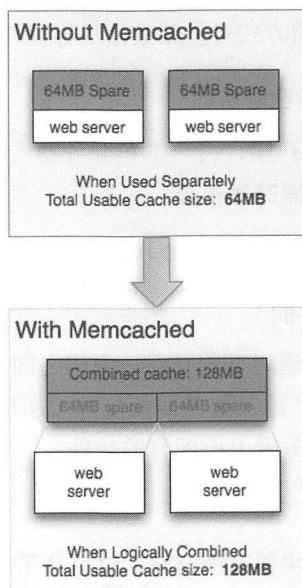


图 5-28 Memcached 的工作原理



- 图上方所示的部署方案（非 Memcached 方案）中的每个节点是完全独立的。
- 图下方所示的部署方案（Memcached 方案）中的每个节点可以利用来自其他节点的内存。

第一种方案是比较常见的部署策略，但是你会发现，它在某种意义上是一种浪费，因为所需缓存的总量只是 Web 应用的实际容量的一小部分，而且需要努力保持所有这些节点上的缓存一致。

而在 Memcached 方案中可以看到，所有的服务器都在使用相同的虚拟内存池。这意味着在整个集群里面，同一个数据总是从相同的位置中存储和检索的。

此外，Memcached 方案也能适应应用程序数据不断增长的需求。在这个示例中，虽然只是简单地用两台 Web 服务器作为演示，但如果有 50 台 Web 服务器，在第一种方案中，仍然只能使用到 64 MB 的可用缓存，但在 Memcached 方案中，能使用到 3.2 GB 的可用缓存。

当然，在实际应用中，一般不会将 Web 服务器作为缓存。许多用户都会在使用 Memcached 时，搭建专门的 Memcached 服务器作为缓存。

Memcached 主要包括以下核心设计理念。

1. 简单的 key-value 存储

Memcached 服务器并不关心数据是什么样子的。数据由一个 key、过期时间、可选的标志及原始数据组成。无须关心数据结构的问题，只需要按预先的序列上传数据即可。可以使用某些命令（incr/decr）来实现以简单的方式对基础数据进行操作。

2. 逻辑一半在客户端，另一半在服务器

Memcached 的实现，一半在客户端，另一半在服务器。用户知道如何选择所要读取或写入的服务器，如何在无法连接到服务器时做出反应。

这些服务器知道如何存储和获取数据，它们还管理着可以被清除或重用的内存。

3. 各个服务器之间的是无连接的

Memcached 服务器之间并不知道对方的存在。它们之间不会有回应，不会同步数据，不会有广播消息，也不会互相复制数据。添加服务器就能够增加可用的存储。缓存失效机制简化了，由客户端来直接删除或覆盖它在服务器上所拥有的数据。

4. O(1)

所有的命令都被设计为尽可能快和对锁友好。这提高了所有用例的确定性查询的速度。

在慢的机器中应在 1 毫秒内执行完查询。高端服务器可以拥有每秒百万次的吞吐量。



5. “遗忘”功能

默认情况下, Memcached 采用了 “Least Recently Used (最近最少使用算法)” 的缓存机制, 数据会在指定的时间后过期。这两者都是很多问题的优雅的解决方案, 既限制了陈旧未使用的数据, 也能保证经常被使用的数据可以被缓存。

垃圾回收器无须 “pauses (暂停)” 等待, 以确保低延迟和懒加载空闲的空间。可以参见 LRU 文档 (网址为 https://github.com/memcached/memcached/blob/master/doc/new_lru.txt) 了解该算法的细节。

6. 缓存失效

无须广播服务器的变化到所有可用的主机上, 作为替代者, 客户端可以直接解决服务器保存的数据失效的问题。

5.5.3 Memcached 客户端

下面介绍常用的 Memcached 客户端的使用方式。

1. telnet

telnet 可以通过命令行的方式来监控 Memcached 服务器存储数据的情况。例如, Memcached 的服务地址为 localhost:11211, 可以 telnet 到该服务端口:

```
$ telnet localhost 11211
```

如果连接成功, 则可以使用如下一些命令。

stats 命令

该命令用于显示服务器信息、统计数据等。下面是各种参数的说明:

STAT pid 22362	//memcache 服务器的进程 ID
STAT uptime 1469315	//服务器已经运行的秒数
STAT time 1339671194	//服务器当前的 UNIX 时间戳
STAT version 1.4.9	//Memcache 版本
STAT libevent 1.4.9-stable	//libevent 版本
STAT pointer_size 64	//当前操作系统的指针大小 (32 位系统一般是 32bit, //64 就是 64 位操作系统)
STAT rusage_user 3695.485200	//进程的累计用户时间
STAT rusage_system 14751.273465	//进程的累计系统时间
STAT curr_connections 69	//服务器当前存储的 items 数量
STAT total_connections 855430	//从服务器启动后存储的 items 总数量
STAT connection_structures 74	//服务器分配的连接构造数



```

STAT reserved_fds 20
STAT cmd_get 328806688 //get 命令（获取）总请求次数
STAT cmd_set 75441133 //set 命令（保存）总请求次数
STAT cmd_flush 34 //flush 命令请求次数
STAT cmd_touch 0 //touch 命令请求次数
STAT get_hits 253547177 //总命中次数
STAT get_misses 75259511 //总未命中次数
STAT delete_misses 4 //delete 命令未命中次数
STAT delete_hits 565730 //delete 命令命中次数
STAT incr_misses 0 //incr 命令未命中次数
STAT incr_hits 0 //incr 命令命中次数
STAT decr_misses 0 //decr 命令未命中次数
STAT decr_hits 0 //decr 命令命中次数
STAT cas_misses 0 //cas 命令未命中次数
STAT cas_hits 0 //cas 命令命中次数
STAT cas_badval 0 //使用擦拭次数
STAT touch_hits 0 //touch 命令未命中次数
STAT touch_misses 0 //touch 命令命中次数
STAT auth_cmds 0 //认证命令处理的次数
STAT auth_errors 0 //认证失败数目
STAT bytes_read 545701515844 //总读取字节数（请求字节数）
STAT bytes_written 1649639749866 //总发送字节数（结果字节数）
STAT limit_maxbytes 2147483648 //分配给 Memcache 的内存大小（字节）
STAT accepting_conns 1 //服务器是否达到过最大连接（0/1）
STAT listen_disabled_num 0 //失效的监听数
STAT threads 4 //当前线程数
STAT conn_yields 14 //连接操作主动放弃数目
STAT hash_power_level 16
STAT hash_bytes 524288
STAT hash_is_expanding 0
STAT expired_unfetched 30705763
STAT evicted_unfetched 0
STAT bytes 61380700 //当前存储占用的字节数
STAT curr_items 28786 //当前存储的数据总数
STAT total_items 75441133 //启动以来存储的数据总数
STAT evictions 0 //为获取空闲内存而删除的 items 数（分配给 Memcache
的空间用满后需要删除旧的 items 来得到空间以分配给新的 items）
STAT reclaimed 39957976 //已过期的数据条目来存储新数据的数目
END

```

stats 命令还有几个二级子项，如表 5-4 所示。

表 5-4 二级子项

命 令	含 义 说 明
stats slabs	表示各个 slab 的信息，包括 chunk 的大小、数目、使用情况等
stats items	显示各个 slab 中 item 的数目和最老 item 的年龄（最后一次访问距离现在的秒数）
stats detail [on off dump]	设置或者显示详细操作记录；参数为 on，打开详细操作记录；参数为 off，关闭详细操作记录；参数为 dump，显示详细操作记录（每一个键值 get、set、hit、del 的次数）
stats malloc	打印内存分配信息
stats sizes	打印缓存使用信息
stats reset	重置统计信息

get 命令

用法：get <key>*\r\n，用于获取缓存的数据，键为 key。示例如下：

```
get name
VALUE name 0 7
shirdrn
END
```

gets 命令

用法：gets <key>*\r\n，用于获取缓存的数据，键为一组 key。示例如下：

```
gets name hobby
VALUE name 1 7
1234567
VALUE hobby 0 25
tenis basketball football
END
```

set 命令

用法：set <key> <flags> <exptime> <bytes> [noreply]\r\n<value>\r\n，向缓存中存储数据，不管 key 对应的值存在与否，都设置 key 对应的值。示例如下：

```
set name 0 1800 7
shirdrn
STORED
get name
```

```
VALUE name 0 7
shirdrn
END
```

touch 命令

用法: touch <key> <exptime> [noreply]\r\n, 更新缓存中 key 对应的值的过期时间。示例如下:

```
touch name 1800
```

delete 命令

用法: delete <key> [<time>] [noreply]\r\n, 给定键 key, 删除缓存中 key 对应的数据。示例如下:

```
delete name 60
```

add 命令

用法: add <key> <flags> <exptime> <bytes> [noreply]\r\n<value>\r\n, 向缓存中存储数据, 只有 key 对应的值不存在时, 才会设置 key 对应的值。示例如下:

```
add hobby 0 1800 10
basketball
STORED
get hobby
VALUE hobby 0 10
basketball
END
```

replace 命令

用法: replace <key> <flags> <exptime> <bytes> [noreply]\r\n<value>\r\n, 覆盖一个已经存在 key 及其对应的 value, 替换一定要保证替换后的值的长度与原始长度相同, 否则 replace 失败。示例如下:

```
get name
VALUE name 0 7
shirdrn
END
replace name 0 1800 7
youak47
STORED
get name
```



```
VALUE name 0 7
youak47
END
```

append 命令

用法: `append <key> <flags> <exptime> <bytes> [noreply]\r\n<value>\r\n`, 在一个已经存在的数据值 (value) 上追加, 在数据值的后面追加。示例如下:

```
get hobby
VALUE hobby 0 10
basketball
END
append hobby 0 1800 9
football
STORED
get hobby
VALUE hobby 0 19
basketball football
END
```

prepend 命令

用法: `prepend <key> <flags> <exptime> <bytes> [noreply]\r\n<value>\r\n`, 在一个已经存在的数据值 (value) 上追加, 是在数据值的前面追加。示例如下:

```
get hobby
VALUE hobby 0 19
basketball football
END
prepend hobby 0 1800 6
tenis
STORED
get hobby
VALUE hobby 0 25
tenis basketball football
END
```

incr 命令

用法: `incr <key> <value> [noreply]\r\n`, 计数命令, 可以在原来已经存在的数字上进行累加求和, 计算并存储新的数值。示例如下:

```
set active_users 0 1000000 7
1000000
STORED
get active_users
VALUE active_users 0 7
1000000
END
incr active_users 99
1000099
```

decr 命令

用法: `decr <key> <value> [noreply]`\r\n，计数命令，可以在原来已经存在的数字上进行减法计算，计算并存储新的数值。示例如下：

```
get active_users
VALUE active_users 0 7
1000099
END
decr active_users 3456
996643
```

flush_all 命令

用法: `flush_all [<time>] [noreply]`\r\n，使缓存中的数据项失效，指可选参数在多少秒后失效。示例如下：

```
flush_all 1800
```

version 命令

用法: `version`\r\n，返回 Memcached 服务器的版本信息。示例如下：

```
version
```

quit 命令

用法: `quit`\r\n，退出 telnet 终端。示例如下：

```
Quit
```

2. Java 客户端

可以使用 Java 语言编写代码来访问 Memcached 缓存。其中，第三方实现的开源客户端较多，现列举一二。

Spymemcached

Spymemcached 是一个采用 Java 开发的异步、单线程的 Memcached 客户端。项目地址为 <https://github.com/dustin/java-memcached-client>。

示例代码如下：

```
import net.spy.memcached.AddrUtil;
import net.spy.memcached.BinaryConnectionFactory;
import net.spy.memcached.MemcachedClient;
import net.spy.memcached.internal.OperationFuture;

public class TestSpymemcached {

    public static void main(String[] args) throws Exception {
        String address = "192.168.4.86:11211";
        MemcachedClient client = new MemcachedClient(new BinaryConnectionFactory(),
            AddrUtil.getAddresses(address));

        String key = "magic_words";
        int exp = 3600;
        String o = "hello";
        // set
        OperationFuture<Boolean> setFuture = client.set(key, exp, o);
        if(setFuture.get()) {
            // get
            System.out.println(client.get(key));

            // append
            client.append(key, " the world!");
            System.out.println(client.get(key));

            // prepend
            client.prepend(key, "Stone, ");
            System.out.println(client.get(key));

            // replace
            o = "This is a test for spymemcached.";
            OperationFuture<Boolean> replaceFuture = client.replace(key,
                exp, o);
```

```
        if(replaceFuture.get()) {
            System.out.println(client.get(key));

            // delete
            client.delete(key);
            System.out.println(client.get(key));
        }
    }

    client.shutdown();
}
```

XMemcached

XMemcached 则是基于另外一款 Java NIO 实现的高性能、可扩展的 Memcached 客户端。项目地址为 <https://github.com/killme2008/xmemcached>。它的主要特点如下：

- 高性能，稳定可靠，已经在众多公司的项目里得到了应用；
- 功能完备：客户端提供分布式、权重、最新最完整的协议支持；
- 可扩展，易于集成；
- 可动态增删 Memcached 节点；
- 客户端操作统计；
- NIO 连接池。

使用方式可以参考官网的示例 <https://github.com/killme2008/xmemcached/wiki/Getting%20started#simple-example>。

3. Node.js 客户端

Memcached 各种语言的客户端代码的逻辑都非常类似，这里对 Node.js 简单举例说明，代码如下：

```
var MemcachedClient = require('memcached');

// 配置 memcached 客户端
var servers = ['192.168.4.86:11211'];
var client = new MemcachedClient(servers);

// 访问 memcached 服务器
```



```
var key = 'ghost';

// set
var value = 'Ghost wind blows!';
client.set(key, 0, value, function(err) {
  var data = 'key=' + key + ', value=' + value;
  if(err) {
    console.error('Fail to set: ' + data);
  } else {
    console.log('Added: ' + data);
  }
});

// get
var valueGot = client.get(key, function(err, data) {
  var dataGot = 'key=' + key + ', valueGot=' + data;
  if(err) {
    console.error('Fail to get: ' + dataGot);
  } else {
    console.log('Got: ' + dataGot);
  }
});
```

5.6 Redis

Redis 是一个 key-value 模型的内存数据存储系统。和 Memcached 类似，它支持存储的 value 类型相对更多，包括 string（字符串）、hash（哈希类型）、list（链表）、set（集合）和 sorted set（有序集合）的 range query（范围查询），以及位图、Hyperlog、空间索引等的 radius query（半径查询）。并具有内置复制、Lua 脚本、LRU 回收、事务及不同级别磁盘持久化功能，同时通过 Redis Sentinel 提供高可用功能，通过 Redis Cluster 提供自动分区。Redis 是完全开源免费的、遵守 BSD 的协议。

5.6.1 Redis 简介

Redis 支持主从同步。可以从主服务器向任意数量的从服务器上同步数据，从服务器可以是关联其他从服务器的主服务器。这使得 Redis 可执行单层树复制。存盘可以有意无意地对数据进行写操作。由于完全实现了发布/订阅机制，使得从数据库在任何地方进行数据同步时，可订

阅一个频道并接收主服务器完整的消息发布记录。同步对读取操作的可扩展性和数据冗余很有帮助。

用户可以在 Redis 数据类型上执行原子操作，比如追加字符串，增加哈希表中的某个值，在列表中增加一个元素，计算集合的交集、并集或差集，获取一个有序集合中最大排名的元素，等等。

为了获取其卓越的性能，Redis 在内存数据集上工作。在内存中，其是否工作取决于用户，如果用户想持久化其数据，则可以通过偶尔转储内存数据集到磁盘上或在一个日志文件中写入每条操作命令来实现。如果用户仅需要一个内存数据库，那么持久化操作可以被选择性禁用。

5.6.2 Redis 的下载与简单使用

下载、解压、编译 Redis 执行：

```
$ wget http://download.redis.io/releases/redis-3.2.3.tar.gz
$ tar xzf redis-3.2.3.tar.gz
$ cd redis-3.2.3
$ make
```

之后，就能在 src 目录下看到这个编译文件了。运行 Redis 执行：

```
$ src/redis-server
```

使用内置的命令行工具来和 Redis 交互：

```
$ src/redis-cli
redis> set foo bar
OK
redis> get foo
"bar"
```

5.6.3 Redis 的数据类型及抽象

Redis 不仅仅是简单的 key-value 存储，实际上还是一个 data structures server（数据结构服务器），用来支持不同的数值类型。在 key-value 中，value 不仅仅局限于 string 类型，它可以是更复杂的数据结构：

- 二进制安全的 string。
- List——一个链表，链表中的元素按照插入顺序排列。
- Set——string 集合，集合中的元素是唯一的，没有排序。

- Sorted set——和 Set 类似，但是每一个 string 元素关联一个浮点数值，这个数值被称为 Score。元素总是通过它们的 Score 进行排序的，所以不像 Set，它可以获取某个范围内的元素（例如获取前 10 个，或者后 10 个）。
- Hash——Hash 就是由关联值的字段构成的 Map。字段和值都是 string。这个与 Ruby 或 Python 的 hash 很类似。
- Bit array（或者简单称为 Bitmap）——像位数值一样通过特别的命令处理字符串：可以设置和清除单独的 bit，统计所有 bit 集合中为 1 的数量，查找第一个设置或者没有设置的 bit，等等。
- HyperLogLogs——这是一个概率统计用的数据结构，可以被用来估计一个集合的基数。

对于所有的例子，我们都使用 redis-cli 工具来演示。这是一个简单但是非常有用的命令行工具，可以用来给 Redis Server 发送命令。

1. Redis key

Redis key 是二进制安全的。这意味着我们可以使用任何二进制序列作为 key，例如从一个像“foo”的字符串到一个 JPEG 文件的内容。空字符串也是一个有效的 key。

关于 key 的一些其他的使用规则：

- 不建议使用非常长的 key。这不仅仅是考虑内存方面的问题，而且在数据集中查找 key 可能需要和多个 key 进行比较。如果当前的任务需要使用一个很大的值，则将它进行 hash 是一个不错的方案（例如，使用 SHA1），尤其是从内存和带宽的角度考虑。
- 非常短的 key 往往也不是一个好主意。如果可以将 key 写成“user:1000:followers”，就不要使用“u1000flw”。首先前者更加具有可读性，其次增加的空间相比 key 对象本身和值对象占用的空间是很小的。当然，短 key 显然会消耗更少的内存，我们需要找到一个适当的平衡点。
- 提倡使用模式。例如，像“user:1000”这样的“object-type:id”模式是一个好主意。点和连接线通常被用在多个单词的字段中，例如，“comment:1234:reply.to”或者“comment:1234:reply-to”。
- 允许 key 最大是 512 MB。

2. Redis String

Redis String 类型是关联到 Redis key 最简单的值类型。它是 Memcached 中唯一的数据类型，所以对于 Redis 新手来说，使用它也是非常自然的。

因为 Redis key 是 String，所以当我们使用 String 类型作为 value 时，其实就是将一个 String

映射到另一个 String。String 数据类型对于大量的用例是非常有用的，例如缓存 HTML 片段或者页面。

让我们一起使用 redis-cli 来操作一下 String 类型：

```
> set mykey somevalue
OK
> get mykey
"somevalue"
```

使用 SET 和 GET 命令可以设置和获取 String 值。注意 SET 会替换已经存入 key 中的任何值，即使这个 key 存在的不是 String 值。所以 SET 执行一次分配。

值可以是任何类型的 String（包括二进制数据），例如可以存一个 JPEG 图片到一个 key 中。但值不能超过 512 MB 大小。

SET 命令有一些有趣的选项，这些选项可以通过额外的参数来设置，例如：

- NX——只在 key 不存在的情况下执行；
- XX——只在 key 存在的情况下执行。

下面是操作示例：

```
> set mykey somevalue
OK
> get mykey
"somevalue"
```

String 是 Redis 的基础值，可以对它们进行一些有意思的操作。例如，进行原子递增：

```
> set counter 100
OK
> incr counter
(integer) 101
> incr counter
(integer) 102
> incrby counter 50
(integer) 152
```

INCR 命令将 String 值解析为 Integer，然后将它递增 1，最后将新值作为返回值。这里也有一些类似的命令，例如 INCRBY、DECR 和 DECRBY。在内部它们是相同的命令，并且执行方式的差别非常微小。

INCR 命令是原子操作，这意味即使多个客户端对同一个 key 发送 INCR 命令也不会导致 Race Condition（竞争条件）问题。例如，当 client1 和 client2 同时给值加 1 时（旧值为 10），

它们不会同时读到 10，最终值一定是 12，因为 read-increment-set 起作用了。

操作 String 有很多的命令。例如 GETSET 命令将一个 key 设置为新值，并将旧值作为返回值。例如，网站接收到新的访问者时，使用 INCR 命令递增一个 Redis key，这时就可以使用 GETSET 命令来实现。如果想每隔一个小时收集一次信息，并且需要不丢失每一次的递增，则可以 GETSET 这个 key，将新值 0 赋给它，并将旧值读回。

MSET 和 MGET 命令用于在一条命令中设置或者获取多个 key 的值，这对减少网络延时是非常有用的。

下面是操作示例：

```
> mset a 10 b 20 c 30
OK
> mget a b c
1) "10"
2) "20"
3) "30"
```

当使用 MGET 时，Redis 返回的是一个值数组。

3. 修改和查询 key 空间

还有一些命令没有定义在具体的类型上，但在与 key 空间交互时非常有用。这些命令可以用于任何类型的 key。

例如，EXISTS 命令返回 1 或者 0 来标志一个给定的 Key 是否在数据库中存在。DEL 命令用来删除一个 key 和关联的值，而不管这个值是什么。

下面是操作示例：

```
> set mykey hello
OK
> exists mykey
(integer) 1
> del mykey
(integer) 1
> exists mykey
(integer) 0
```

通过这个例子，也可以看到 DEL 命令根据 key 是否被删除返回 1 或者 0。

这里有很多与 key 空间相关的命令，但是上面两个命令及 TYPE 命令是非常关键的命令。TYPE 命令返回在指定的 key 中存放的值的类型。

下面是操作示例：

```
> set mykey x
OK
> type mykey
string
> del mykey
(integer) 1
> type mykey
none
```

4. Redis 失效：具有有限生存时间的 key

Redis 失效是 Redis 的一个特性之一。这个特性可以用在任何一种值类型中。可以给一个 key 设置一个超时时间，这个超时时间就是有限的生存时间。当这个生存时间过去时，这个 key 会自动被销毁。

下面是一些关于 Redis 失效的描述：

- 在设置失效时间时，可以使用秒或者毫秒精度。
- 失效时间一般总是 1ms。
- 失效信息会被复制，并持久化到磁盘中。当 Redis 服务器停止时（这意味着 Redis 将保存 key 的失效时间），这个时间会在无形中度过。

设置失效时间是轻而易举的：

```
> set key some-value
OK
> expire key 5
(integer) 1
> get key (immediately)
"some-value"
> get key (after some time)
(nil)
```

这个 key 在两次 GET 调用之间消失了，因为第二次调用延时超过了 5 秒。在上面的例子中，我们使用 EXPIRE 命令来设置超时时间（它也可以用来给一个已经设置超时时间的 key 设置一个不同的值。PERSIST 可以用来删除失效时间，并将 key 永远持久化）。当然我们也可以使用其他 Redis 命令来创建带失效时间的 key。例如，使用 SET 选项：

```
> set key 100 ex 10
OK
> ttl key
```

```
(integer) 9
```

上面的例子设置 key 值为 String 100，并带有 10 秒的超时时间。之后，使用 TTL 命令检测这个 key 的剩余生存时间。

如果想知道如何以毫秒级设置和检测超时时间，则可以查看 PEXPIRE 和 PTTL 命令，以及 SET 选项列表，则可以参见 <http://redis.io/commands>。

5. Redis List

Redis List 是通过 Linked List 实现的。这意味着即使成千上万的元素在一个列表中，在列表头和尾增加一个元素的操作是在一个常量时间内完成的。使用 LPUSH 命令增加一个新元素到一个具有 10 个元素的列表头的速度和增加一个元素到有千万元素的列表头是一样的。

这样做的负面影响是什么呢？在使用数组实现的列表中，使用 index 访问一个元素是非常快的（index 访问是常量时间），而在使用 Linked List 实现的 List 中不是那么快的（这个操作需要的工作量和被访问元素的 index 成正比）。

Redis List 使用 Linked List 实现的原因对于数据库系统而言，能够快速增加一个元素到一个非常长的列表中是非常关键的。

下面你将会看到，Redis List 的另一个重要优势是可以在常量时间内获取一个固定长度的子 List。

如何实现快速访问一个庞大元素集合的中间值？可以使用另一个数据结构，它叫作 Sorted Set。

6. 使用 Redis List 的第一步

LPUSH 命令将一个新元素从左边加入列表，而 RPUSH 命令将一个新元素从右边加入列表。最后，LRANGE 命令获取列表范围内的元素：

```
> rpush mylist A
(integer) 1
> rpush mylist B
(integer) 2
> lpush mylist first
(integer) 3
> lrange mylist 0 -1
1) "first"
2) "A"
3) "B"
```

注意 LRANGE 带有两个 index，分别是返回范围的开始和结束。这两个 index 都可以是负

数，告诉 Redis 从后边开始计数：-1 表示最后一个元素，-2 表示倒数第二个元素，以此类推。

正如你看到的，RPUSH 将元素附加到列表右边，而 LPUSH 将元素附加到左边。

两个命令都是 variadic commands（可变参数的命令）。这意味着可以在一次调用中将多个元素插入列表中。

下面是操作示例：

```
> rpush mylist 1 2 3 4 5 "foo bar"
(integer) 9
> lrange mylist 0 -1
1) "first"
2) "A"
3) "B"
4) "1"
5) "2"
6) "3"
7) "4"
8) "5"
9) "foo bar"
```

Redis List 的一个重要操作是 pop 元素的能力。pop 元素是指从列表中取出元素，并同时将它从列表中删除的操作。可以从左边或者右边 pop 元素，这和从列表的两侧 push 元素是类似的。

下面是操作示例：

```
> rpush mylist a b c
(integer) 3
> rpop mylist
"c"
> rpop mylist
"b"
> rpop mylist
"a"
```

我们加入了三个元素，并 pop 了三个元素，所以在这些命令执行完后，这个列表是空的，并且没有更多的元素可以 pop。如果尝试再 pop 一个元素，则下面就是我们获得的结果：

```
> rpop mylist
(nil)
```

Redis 返回 NULL 值来表示已经没有元素在列表中了。

7. List 常见的用例

List 对于某些特定的场景是非常有用的。两个非常典型用例如下：

- 记录用户 post 到社区网络的最新更新。
- 使用消费者—生产者模式进行进程间通信。生产者推送数据到 List 中，消费者消费这些数据并执行操作。Redis 有专门的 List 命令使这个用例更加可靠和高效。

例如，Ruby 库 `resque` 和 `sidekiq` 在底层就是使用 Redis 列表实现的后台任务。Twitter 社交网络将用户最新 post 的 tweet 放入 Redis List 中。

为了一步步描述这个常见的用例，设想在主页上展示来自社交网站上发布的最新图片，并且想提高访问速度。

- 每次一个用户 post 一张新的图片，我们就使用 `LPUSH` 将它的 ID 加入一个 List 中。
- 当用户访问这个主页时，我们使用 `LRANGE 0 9` 来获得最近上传的 10 个数据。

8. Capped list（限制列表）

在很多的用例中，我们仅需要使用 List 保存最近的元素，比如，社交网络的更新、日志或其他任何事。

Redis 允许我们使用 List 作为 capped 集合，使用 `LTRIM` 命令来仅记住最近的 N 个元素，并丢失所有的旧数据。

`LTRIM` 命令和 `LRANGE` 类似，但它设置这个范围作为新的 List 值，而不是展示指定范围的元素。所有在给定范围之外的元素都会被删除。

通过下面的例子，我们可以使它更加容易理解：

```
> rpush mylist 1 2 3 4 5
(integer) 5
> ltrim mylist 0 2
OK
> lrange mylist 0 -1
1) "1"
2) "2"
3) "3"
```

上面的 `LTRIM` 命令告诉 Redis 仅取从 index 0~2 的列表元素，其他的会被丢弃。这允许一个非常简单但又很有用的模式：执行一个 List 推送操作和一个 List 截断操作，来增加一个新元素，并丢弃超过限制的元素。用法如下：

```
LPUSH mylist <some element>
LTRIM mylist 0 999
```



上面的组合增加了一个新元素，并取 1000 个最新的元素放入这个 List 中。通 LRANGE 命令，可以访问最前面的数据，而不需要记住非常旧的数据。

注意：LRANGE 是一个 $O(N)$ 的命令，访问向列表头或者尾的小范围元素是一个常量时间操作。

9. List 上的阻塞操作

List 有一个特别的特性，非常适合用来实现队列，并且它也是内部进程通信系统的一个基本构件：阻塞操作。

设想使用一个进程将数据推送到一个列表中，并且使用一个不同的进程在这些数据上做一些工作。这是一个普通的生产者—消费者模式，并且可以通过下面简单的方法来实现。

- 为了推送数据到这个 List 中，生产者调用 LPUSH 命令。
- 为了从列表中提取/处理数据，消费者调用 RPOP 命令。

然而，有时这个列表可能是空的，没有什么需要处理的，所以 RPOP 仅返回 NULL。在这种情况下，消费者强制等待一些时间，并使用 RPOP 进行重试。这种方式被称为 polling（拉），在这个场景下并不很合适，因为它有下面几个缺点：

- 迫使 Redis 和客户端处理无用的命令（所有的请求在 List 为空的时候没有实际工作要做，它们仅是返回 NULL）；
- 增加了数据处理的延时，因为在一个工作者接收到 NULL 之后，它等待了一段时间。为了使延时更小，我们可以在调用 RPOP 之间等待更短时间，但负面影响是放大了上面的第一个问题，因为这意味着产生了更多无用的 Redis 调用。

所以 Redis 实现了 BRPOP 和 BLPOP 命令。它们是 RPOP 和 LPOP 的另一个版本，可以在 List 为空的时候进行阻塞，仅当一个新的元素加入列表，或者用户指定的超时时间到达时，它们才会返回给调用者。

下面是一个我们可以在工作者中调用 BRPOP 的例子：

```
> brpop tasks 5
1) "tasks"
2) "do_something"
```

它的意思是“等待在列表 tasks 中的元素，但是如果 5 秒钟过后还没有元素有效则返回”。

注意：可以使用 0 作为超时时间来表示永久等待元素，并且也可以指定多个 List 而不仅是一个。这样就可以在同一时间等待多个 List，并且当第一个 List 接收到元素时获得通知。

关于 BRPOP 的一些注意事项。

- 客户端按照顺序被服务：当某个客户端推送一个元素时，第一个阻塞等待 List 的客户端



会被最先服务。

- BRPOP 的返回值和 RPOP 是不同的：它是两个元素的数组。数组中除了新加入的元素，还有 key 的名称，这是因为 BRPOP 和 BLPOP 能够等待多个队列中的元素。
- 如果超时时间到，则返回 NULL。

关于 List 和阻塞操作，建议阅读下面内容以获取更多的信息。

- 可以使用 RPOPLPUSH 构建安全队列或者循环队列（网址为 <http://redis.io/commands/rpoplpush>）。
- 还有一个这个命令的阻塞版本，称为 BRPOPLPUSH（网址为 <http://redis.io/commands/brpoplpush>）。

10. key 的自动创建和删除

截至目前，在我们的例子中，从没有在推送元素之前必须创建空列表，或者里面不再有元素时删除空列表。当 List 为空时，Redis 会负责删除这个 key；当 key 不存在时，而我们正在尝试增加一个新元素给它，Redis 会负责创建一个空列表。

这不是 List 特有的，它对所有包含多个元素的 Redis 数据类型都有效，包括 Set、Sorted Set 和 Hash。

可以用下面的三个规则来总结这个行为：

- 当我们增加一个元素到一个聚合数据类型（aggregate data type）中时，如果目标 key 不存在，那么在增加元素之前会自动创建一个空的聚合数据类型。
- 当我们从一个聚合数据类型中删除元素时，如果值为空，那么这个 key 会自动销毁。
- 对于一个空 key，调用一个只读命令，如 LLEN（返回列表的长度），或者调用一个删除元素的写命令，总是和持有空聚合类型的 key 产生一样的结果。

第 1 个规则的例子：

```
> del mylist
(integer) 1
> lpush mylist 1 2 3
(integer) 3
```

然而，我们不能对 key 中存在的错误类型执行操作：

```
> set foo bar
OK
> lpush foo 1 2 3
(error) WRONGTYPE Operation against a key holding the wrong kind of value
```



```
> type foo
string
```

第2个规则的例子：

```
> lpush mylist 1 2 3
(integer) 3
> exists mylist
(integer) 1
> lpop mylist
"3"
> lpop mylist
"2"
> lpop mylist
"1"
> exists mylist
(integer) 0
```

在所有的元素 pop 之后，这个 key 不再存在。

第3个规则的例子：

```
> del mylist
(integer) 0
> llen mylist
(integer) 0
> lpop mylist
(nil)
```

11. Redis Hash

Redis hash 是 field-value 对：

```
> hmset user:1000 username antirez birthyear 1977 verified 1
OK
> hget user:1000 username
"antirez"
> hget user:1000 birthyear
"1977"
> hgetall user:1000
1) "username"
2) "antirez"
```




```
3) "birthyear"  
4) "1977"  
5) "verified"  
6) "1"
```

Hash 可以非常方便地表示对象，事实上能放入 Hash 字段的数量是没有限制的（除了可用内存），所以可以在应用中以不同的方式使用 Hash。

HMSET 可以用来设置 Hash 的多个字段，HGET 获取单个字段。HMGET 和 HGET 类似，区别在于前者返回的是一个多个字段对应值的数组：

```
> hmget user:1000 username birthyear no-such-field  
1) "antirez"  
2) "1977"  
3) (nil)
```

这里也有能够在单个字段上执行操作的命令，例如 HINCRBY：

```
> hincrby user:1000 birthyear 10  
(integer) 1987  
> hincrby user:1000 birthyear 10  
(integer) 1997
```

可以在文档（<http://redis.io/commands#hash>）中找到 Hash 命令的完整列表。

值得注意的是，小 Hash（例如，一些带有小值的元素）在内存中使用特殊方式编码，这样使它们的内存使用非常高效。

12. Redis Set

Redis Set 是 String 的无序集合。SADD 命令增加新元素到一个 Set 中。还有一些针对 Set 的其他操作，例如检查一个元素是否已经存在，可以在多个 Set 上执行交、并或差运算等。

```
> sadd myset 1 2 3  
(integer) 3  
> smembers myset  
1. 3  
2. 1  
3. 2
```

这里，我们加入三个元素到 Set 中，并且告诉 Redis 返回所有元素。正如所看到的，它们是无序的，在每次调用时，Redis 以任意顺序返回元素，没有和用户有任何关于元素顺序的约定。

Redis 有用来测试成员关系的命令。执行：



```
> sismember myset 3
(integer) 1
> sismember myset 30
(integer) 0
```

“3”是 Set 的元素，而“30”不是。

Set 善于用来表达对象间的关系。例如，我们可以很容易用 Set 实现打标签。

我们用想被打标签的每一个对象来构建一个 Set。这个 Set 包含关联这个对象的标签（tag）ID。

如果新闻（news）ID 使用 1、2、5 和 77 来打标签，那么就可以使用一个 Set 关联标签 ID 和新闻数据。

```
> sadd news:1000:tags 1 2 5 77
(integer) 4
```

有时可能也有反向关系：所有新闻列表都使用一个给定的 tag 来打标签。

```
> sadd tag:1:news 1000
(integer) 1
> sadd tag:2:news 1000
(integer) 1
> sadd tag:5:news 1000
(integer) 1
> sadd tag:77:news 1000
(integer) 1
```

获得一个给定对象的所有 tag 是非常简单的：

```
> smembers news:1000:tags
1. 5
2. 1
3. 77
4. 2
```

注意：在这个例子中，我们是用一个 Redis Hash 来将 tag ID 映射到 tag 名称上的。

这里也有其他一些比较复杂的操作，仍可以很简单地使用恰当的 Redis 命令来实现。例如，我们可能想得到所有带有 tag 1, 2, 10 和 27 的对象列表。我们可以使用 SINTER 命令来做到这一点，这个命令可以在不同的 Set 上做交集。可以使用：

```
> sinter tag:1:news tag:2:news tag:10:news tag:27:news
... results here ...
```



处理交集操作，也可以执行并、差或提取一个随机元素等操作。

提取元素的命令是 SPOP，非常适合这种问题的建模。例如，为了实现一个 Web 的扑克牌游戏，你可能想使用 Set 来表示。设想我们使用一个字母前缀表示 4 种花色的扑克牌 (C)lubs、(D)iamonds、(H)earts、(S)pades:

```
> sadd deck C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 CJ CQ CK
D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 DJ DQ DK H1 H2 H3
H4 H5 H6 H7 H8 H9 H10 HJ HQ HK S1 S2 S3 S4 S5 S6
S7 S8 S9 S10 SJ SQ SK
(integer) 52
```

现在我们想提供给每一个玩家 5 张牌。使用 SPOP 命令删除一个随机元素，并将它返回给客户端，所以在这种情况下，这是完美的操作。

如果对我们的牌直接调用它，下一次玩这个游戏的时候，我们需要再次获得一副牌。这可能就不是完美的。所以在开始的时候，我们可以复制 deck key 中的 Set 到 game:1:deck 的 key 中。

可以使用 SUNIONSTORE 来完成。这个命令是执行多个 Set 的并操作，并保存结果到另一个 Set 中。然而，一个单独 Set 的并操作是它自己，我们可以通过下面的操作来复制 deck:

```
> sunionstore game:1:deck deck
(integer) 52
```

现在，我们已经可以提供给第一个玩家 5 张牌:

```
> spop game:1:deck
"C6"
> spop game:1:deck
"CQ"
> spop game:1:deck
"D1"
> spop game:1:deck
"CJ"
> spop game:1:deck
"SJ"
```

现在介绍获取 Set 内元素数量命令。这个命令通常被称为一个 Set 的基数，在 Set 理论上下文中，对应的 Redis 命令被称为 SCARD。

```
> scard game:1:deck
(integer) 47
```



数学计算： $52 - 5 = 47$ 。

当你仅需要获取随机元素，而不需要从 Set 中删除它们时，可以使用 SRANDMEMBER 命令。它也提供返回重复和非重复元素的功能。

13. Redis Sorted set

Sorted Set 是和 Set 与 Hash 的混合体很类似的一种数据类型。和 Set 一样，Sorted Set 由唯一、不重复的 String 元素组成。所以在某种意义上，Sorted Set 也是 Set。

然而，Set 中的元素是无序的，而 Sorted Set 中每一个元素关联一个被称为分数（Score）的浮点值（这就是为什么这个类型也和 Hash 类似，因为每一个元素都映射到一个值）。

此外，还可以有序获取 Sorted Set 中的元素（所以不需要明确要求 Sorted Set 进行排序，排序是 Sorted Set 的特性）。它们根据下面的规则排序：

- A 和 B 是具有不同分数的元素，如果 $A.score > B.score$ ，则 $A > B$ 。
- A 和 B 确实有相同的分数，如果 A 的 String 比 B 的字典顺序大，则 $A > B$ 。A 和 B 的 String 是不可能相等的，因为 Sorted Set 不会包含重复的元素。

让我们从一个简单的例子开始，将一些选定的黑客名称作为元素增加到 Sorted Set 中，它们的生日年份作为 Score。

```
> zadd hackers 1940 "Alan Kay"
(integer) 1
> zadd hackers 1957 "Sophie Wilson"
(integer) 1
> zadd hackers 1953 "Richard Stallman"
(integer) 1
> zadd hackers 1949 "Anita Borg"
(integer) 1
> zadd hackers 1965 "Yukihiro Matsumoto"
(integer) 1
> zadd hackers 1914 "Hedy Lamarr"
(integer) 1
> zadd hackers 1916 "Claude Shannon"
(integer) 1
> zadd hackers 1969 "Linus Torvalds"
(integer) 1
> zadd hackers 1912 "Alan Turing"
(integer) 1
```




正如所看到的，ZADD 和 SADD 类似，但是带有一个额外的参数（放置在被增加元素的前面），这就是 Score。ZADD 也是可变列表（Variadic），所以可以随意指定多个 score-value 对，即使上面的例子中没有使用。

实现注意事项：Sorted Set 通过一个 dual-ported 数据结构，包含一个 skip 列表和一个 Hash 表，所以每次我们增加一个元素，Redis 都执行一个 $O(\log(N))$ 的操作。这是非常好的实践，因为当我们请求排序的元素时，Redis 完全不需要做任何工作，所有的元素都已经排序。

```
> zrange hackers 0 -1
1) "Alan Turing"
2) "Hedy Lamarr"
3) "Claude Shannon"
4) "Alan Kay"
5) "Anita Borg"
6) "Richard Stallman"
7) "Sophie Wilson"
8) "Yukihiro Matsumoto"
9) "Linus Torvalds"
```

注意：0 和 -1 表示元素 index 0 到最后一个元素（-1 在这里和 LRange 命令中的 index 规则是一样的）。

如果想反向排序它们，则使用 ZREVRANGE 代替 ZRANGE：

```
> zrevrange hackers 0 -1
1) "Linus Torvalds"
2) "Yukihiro Matsumoto"
3) "Sophie Wilson"
4) "Richard Stallman"
5) "Anita Borg"
6) "Alan Kay"
7) "Claude Shannon"
8) "Hedy Lamarr"
9) "Alan Turing"
```

我们也可以返回 Score，使用 WITHSCORES 参数：

```
> zrange hackers 0 -1 withscores
1) "Alan Turing"
2) "1912"
3) "Hedy Lamarr"
4) "1914"
```



- 5) "Claude Shannon"
- 6) "1916"
- 7) "Alan Kay"
- 8) "1940"
- 9) "Anita Borg"
- 10) "1949"
- 11) "Richard Stallman"
- 12) "1953"
- 13) "Sophie Wilson"
- 14) "1957"
- 15) "Yukihiro Matsumoto"
- 16) "1965"
- 17) "Linus Torvalds"
- 18) "1969"

14. 在范围上的操作

Sorted Set 比上面提到的更强大。它们可以对一个范围进行操作。让我们获得所有 1950 年之前出生的人的数据。可以使用 ZRANGEBYSCORE 命令来做到这一点：

```
> zrangebyscore hackers -inf 1950
1) "Alan Turing"
2) "Hedy Lamarr"
3) "Claude Shannon"
4) "Alan Kay"
5) "Anita Borg"
```

我们要求 Redis 返回所有从负无穷到 1950 的 Score 的所有元素（两个极端都包含）。

也可以删除一个区域的所有元素。让我们从 Sorted Set 中删除从 1940~1960 年出生的所有黑客：

```
> zremrangebyscore hackers 1940 1960
(integer) 4
```

ZREMRANGEBYSCORE 可能不是最好的命令名称，但是它非常有用。它的返回值是删除元素的数量。

Sorted Set 的另一个非常有用的命令是获取位置操作。可以用来查询一个元素在 Sorted Set 中的排序位置。

```
> zrank hackers "Anita Borg"
(integer) 4
```



ZREVRANK 命令也可以获得元素位置，但是是元素反序排列的位置。

15. Lexicographical (字典顺序) scores

Redis 2.8 版本中引入了一个新的特性——获取字典顺序 (Lexicographical) 范围。可以设想这个特性就是在 Sorted Set 中所有元素插入时带有相同的 Score。其中，元素使用 C 语言的 memcmp 函数进行比较，所以保证没有冲突，并且每一个 Redis 实例回复相同的输出。

字典顺序范围操作的主要命令有 ZRANGEBYLEX、ZREVRANGEBYLEX、ZREMRANGEBYLEX 和 ZLEXCOUNT。

例如，让我们再次加入著名的黑客列表，但是这次对所有元素使用 Score 0。

```
> zadd hackers 0 "Alan Kay" 0 "Sophie Wilson" 0 "Richard Stallman" 0
  "Anita Borg" 0 "Yukihiro Matsumoto" 0 "Hedy Lamarr" 0 "Claude Shannon"
  0 "Linus Torvalds" 0 "Alan Turing"
```

根据 Sorted Set 的排序规则，它们将以字典顺序排序：

```
> zrange hackers 0 -1
1) "Alan Kay"
2) "Alan Turing"
3) "Anita Borg"
4) "Claude Shannon"
5) "Hedy Lamarr"
6) "Linus Torvalds"
7) "Richard Stallman"
8) "Sophie Wilson"
9) "Yukihiro Matsumoto"
```

通过使用 ZRANGEBYLEX，我们可以得到字典顺序范围：

```
> zrangebylex hackers [B [P
1) "Claude Shannon"
2) "Hedy Lamarr"
3) "Linus Torvalds"
```

范围可以设置为“包含”或“排除”规则（根据第一个字符）。string infinite 和 minus infinite 分别使用“+”和“-”来指定。

这个特性很重要，因为它允许 Sorted set 使用通用的 index。例如，如果通过一个 128-bit 无符号整型参数来“index”一个元素，则可以将 Sorted Set 中的元素设置成相同的 Score（比如 0），但在元素前面加上一个 8 byte 前缀。这个 8 byte 由 128 bit 数字以大端字节序 (big endian) 组成。

因为数字是大端字节序，所以当按照字典顺序排序时，事实上也是按照数字排序的。我们可以获取在 128bit 空间中的任何范围，并且获得元素的值，然后抛弃这个前缀。

如果想在更加正式的 demo 中查看这个特性，则可以参考 Redis autocomplete demo（网址为 <http://autocomplete.redis.io/>）。

16. 更新 Score：积分榜

Sorted Set 的分数在任何时候都可以被修改。仅需要在 Sorted Set 中包含的元素上调用 ZADD 命令，就可以更新它的值和位置。其时间复杂度是 $O(\log(N))$ 。正是因为这样，Sorted Set 也适用于有大量更新的场景。

正因为这个特性，一个常见的用例是积分榜。Facebook 游戏就是一个经典的应用。通过结合按最高分排序用户的能力和获取位置的操作，来显示最前 N 位用户，以及在积分榜上的用户位置。

17. Bitmap（位图）

Bitmap 不是一个真实的数据类型，而是定义在 String 类型上面的面向 bit 操作的集合。因为 String 是二进制安全的，所以最大长度可以是 512 MB。它们适合设置 2^{32} 个不同的 bit。

bit 操作被分成两个组：

- 常量时间单 bit 操作，例如设置一个 bit 为 1 或 0，或者获得它的值；
- 在 bit 组上的操作，例如，对一个给定 bit 范围的 bit 的计数（例如，人口统计）。

Bitmap 的最大优点之一是在存储信息时，它们经常能节约大量的空间。例如，在一个系统中，不同用户通过递增用户 ID 来表示。可以记录单个 bit 的信息（例如，仅需要 512 MB 内存就可以记录 40 亿用户中每一个用户是否接受一个刊物）。

使用 SETBIT 和 GETBIT 命令设置或获取 bit：

```
> setbit key 10 1
(integer) 1
> getbit key 10
(integer) 1
> getbit key 11
(integer) 0
```

SETBIT 命令将 bit 序号作为它的第一个参数，需要设置的值作为它的第二个参数。

GETBIG 仅返回在指定位置上的 bit 值。在范围以外的 bit（定位一个在目标 key 保存的 string 的长度之外的 bit）总是被认为是 0。



在 bit 组上的三个命令操作：

- BITOP 在不同的 string 按位来操作；提供的操作有 AND、OR、XOR 和 NOT；
- BITCOUNT 获取设置成 1 的 bit 数量；
- BITPOS 查找指定 0 或 1 值的第一个 bit。

BITOPS 和 BITCOUNT 能够在 string 的 byte 范围上操作，而不是在整个 string 长度上运行。下面是一个 BITCOUNT 调用的简单例子。

```
> setbit key 0 1
(integer) 0
> setbit key 100 1
(integer) 0
> bitcount key
(integer) 2
```

Bitmap 的常见用例：

- 所有类型的实时分析；
- 高效的存储空间，同时具有高性能的关联对象 ID 的布尔信息。

例如，想知道网站用户日访问最长的 streak。从网站开放上线开始，自 0 开始统计天数，每次有用户访问网站就使用 SETBIT 设置一个 bit。使用当前 UNIX 时间戳作为 bit 的 index，减去初始偏移量，并除以 3600 乘以 24。

对于每一个用户，使用一个小 string 包含每天的访问信息。使用 BITCOUNT 可以很容易得到一个用户访问网站的天数。使用一些 BITPOS 调用简单的获取和分析 bitmap 客户端，就能很容易地计算出最长的 streak。

Bitmap 可以很容易地分割成多个 key。因为需要将数据集合分片，以及避免使用巨大的 key，我们将一个 Bitmap 分割成不同的 key，而不是将所有的 bit 设置到一个 key 中。实现这个的一个简单策略是每个 key 仅保存 M 个 bit，并使用 $\text{bit-number}/M$ 获得 key 的名称，第 N 个 bit 通过 $\text{bit-number} \bmod M$ 来在 key 中定位。

18. HyperLogLogs

HyperLogLog 可以接收多个元素作为输入，并给出输入元素的基数估算值。

- **基数：**集合中不同元素的数量。比如 {'apple', 'banana', 'cherry', 'banana', 'apple'} 的基数就是 3。
- **估算值：**算法给出的基数并不是精确的，可能会比实际稍微多一些或稍微少一些，但会控制在合理的范围之内。



HyperLogLog 的优点是即使输入元素的数量或体积非常大，计算基数所需的空间总是固定的，并且是很小的。在 Redis 里面，每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^{64} 个不同元素的基数。这和计算基数时元素越多耗费内存就越多的集合形成鲜明对比。但是，因为 HyperLogLog 只会根据输入元素来计算基数，而不会储存输入元素本身，所以 HyperLogLog 不能像集合那样返回输入的各个元素。

在 Redis 中，HLLs (HyperLogLog) 是不同的数据结构，它被强制看作 Redis string，所以可以用 GET 来序列化一个 HLL，使用 SET 来反序列化它到服务器上。

从概念上讲，HLL API 像使用 Set 一样来做相同的任务。使用 SCARD 来检查 Set 中元素的数量，它是唯一的，因为 SADD 不会重复添加已经存在的元素。

HLL API 用法类似如下：

- 每次看到一个新元素，都使用 PFADD 把它添加到计数器中。
- 每次想获取当前唯一元素的近似位置，都使用 PFADD，而后使用 PFCOUNT。

```
> pfadd hll a b c d
(integer) 1
> pfcount hll
(integer) 4
```

使用这个数据结构的一个例子是网站每天访问的独立 ID 的数量。

5.7 MongoDB

与 Redis 或 HBase 不同，MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富、最像关系数据库的，旨在为 Web 应用提供可扩展的高性能数据存储解决方案。它支持的数据结构非常松散，类似 JSON 的 BSON 格式，因此可以存储比较复杂的数据类型。MongoDB 最大的特点是其支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且支持对数据建立索引。

5.7.1 MongoDB 简介

MongoDB 的文档结构如图 5-29 所示。



```

{
  name: "sue",           ← field: value
  age: 26,               ← field: value
  status: "A",           ← field: value
  groups: [ "news", "sports" ] ← field: value
}

```

图 5-29 MongoDB 的文档结构

使用文档的优点是：

- 文档（即对象）在许多编程语言里，可以对应于原生数据类型。
- 嵌入式文档和数组可以减少昂贵的连接操作。
- 动态模式支持流畅的多态性。

5.7.2 MongoDB 核心概念

1. 数据库和集合

MongoDB 存储 BSON 文档（即数据记录）在集合（collection）里面，而集合在数据库（database）中。

MongoDB 的集合示意图如图 5-30 所示。

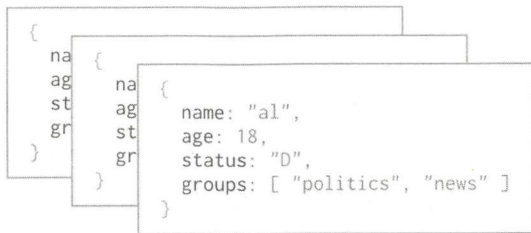


图 5-30 MongoDB 的集合示意图

在 MongoDB 中，数据库用于保存文档的集合。

选择要使用的数据库，使用 mongo shell 的 `use <db>` 语句，示例如下：

```
use myDB
```

创建数据库

如果数据库不存在，则可以在第一次使用数据库时创建数据库。因此，我们可以切换到一个不存在的数据库，而后使用 mongo shell 执行以下操作：

```
use myNewDB
```



```
db.myNewCollection1.insert( { x: 1 } )
```

insert()操作会同时创建数据库 myNewDB 和集合 myNewCollection1（如果它们都不存在）。

MongoDB 存储文档在集合中，集合类似于关系数据库中的表。

创建集合

如果集合不存在，则 MongoDB 会在第一次存储集合数据时创建集合。

```
db.myNewCollection2.insert( { x: 1 } )
db.myNewCollection3.createIndex( { y: 1 } )
```

无论是 insert()，还是 createIndex()操作，都会在集合不存在时创建各自的集合。

数据库在命名时有一定的限制，详细内容可以参阅 <https://docs.mongodb.com/manual/reference/limits/#restrictions-on-collection-names>。

显示创建

MongoDB 中提供了 db.createCollection()方法来显式地创建带有各种选项的集合，比如设置最大尺寸或文档验证规则。如果并不需要显式地创建集合，则可以不指定这些选项，因为 MongoDB 在首次存储集合数据时已创建了新的集合。

要修改这些集合选项，请参阅 collMod（网址为 <https://docs.mongodb.com/manual/reference/command/collMod>）。

文档验证

在默认情况下，集合并不要求其文件具有相同的模式，即在一个单一集合中，文件可以不必具有相同的字段集，且字段的数据类型可以在跨文档的同一个集合中不同。

自 MongoDB 3.2 起，我们可以在更新和插入操作执行中强制使用集合文档验证规则。详细信息请参见文档验证（<https://docs.mongodb.com/manual/core/document-validation/>）。

修改文档结构

要改变集合中文档的结构，例如添加新的字段、删除现有的字段或更改字段值到一个新的类型中，就要更新文档到新的结构。

2. Capped Collection（限制集合）

Capped Collection（限制集合）是固定大小的集合，用于支持基于文档插入顺序的高吞吐率的插入和检索操作。Capped Collection 的工作原理在某种程度上类似于 circular buffer（循环缓冲区）：一旦一个文档填满分配给它的空间，它就将通过在 Capped Collection 中重写老文档来给新文档让出空间。



插入顺序

Capped Collection 能够保留插入顺序。因此，查询是按照文档的插入顺序而不是使用索引来确定插入位置的，这样可以提高增添数据的效率，所以 Capped Collection 可以支持更好的插入吞吐率。

最旧文档的自动删除

为了为新文档腾出空间，在不需要脚本或显式删除操作的前提下，Capped Collection 会自动删除集合中最旧的文档。

举例来说，MongoDB 的操作日志文件 oplog.rs 就是利用 Capped Collection 来实现的。

- 存储高容量系统生成的日志信息。在没有索引的情况下向限制集中插入文档的速度接近于直接在文件系统中写日志的速度。此外，内建的 first-in-first-out（先进先出）特性在管理存储使用时维护了事件的顺序。
- 在 Capped Collection 中缓存少量的数据。因为缓存是读远大于写的，因此或者确保集合经常驻留在工作集（即运行内存）中，或者接收一些需要索引的 write penalty（写惩罚）。

_id 索引

Capped Collection 有一个 _id 字段，并且默认在 _id 字段上创建索引。

更新

如果打算更新 Capped Collection 中的文档，则创建一个索引就可以保证这些更新操作不需要进行集合扫描。

文档大小

在 MongoDB 3.2 版之后，如果一个更新或换操作改变了文档大小，则操作将会失败。

文档删除

不能从一个 Capped Collection 中删除文档，为了从一个集合中删除所有文档，使用 drop() 方法删除集合，然后重新创建 Capped Collection。

分片 (Sharding)

不能对 Capped Collection 进行分片。

查询效率

用自然顺序检索集合中大部分最近插入的元素。这类似于在查询日志文件的尾部内容。

聚合 \$out

聚合管道操作器 \$out 不能将结果写入 Capped Collection。



创建 Capped Collection

必须使用 `db.createCollection()` 方法显式创建 Capped Collection，在 mongo shell 的 `create` 命令中可以查看帮助信息。当创建 Capped Collection 时，必须指定以字节为单位的最大集合大小，而 MongoDB 将预先分配集合。Capped Collection 的大小包括内部消耗的一小部分空间。

```
db.createCollection( "log", { capped: true, size: 100000 } )
```

如果 `size` 字段小于或等于 4096，则该集合将会有 4096 个字节。否则 MongoDB 将会在给定大小的基础上增加为 256 的整数倍。

另外，我们可以为集合指定最大文档数量，使用 `max` 字段，用法如下：

```
db.createCollection("log", { capped : true, size : 5242880, max : 5000 } )
```

`size` 参数始终是必需的，即使指定了文件的 `max` 数量。如果集合达到最大数量的限制，则在没有达到最大文档计数之前，MongoDB 将删除旧文档。

查询 Capped Collection

如果在 Capped Collection 上执行一个没有指定排序的 `find()` 方法，MongoDB 将保证结果的顺序和插入顺序是相同的。

如果想实现用同插入相反的顺序来检索文档，则使用 `find()` 连同 `sort()` 的方法，将 `$natural` 参数设置为 -1，就像下面的例子：

```
db.cappedCollection.find().sort( { $natural: -1 } )
```

检查一个集合是否是 Capped Collection

用 `isCapped()` 方法判定一个集合是否是 Capped Collection，如下所示：

```
db.collection.isCapped()
```

将集合转换为 Capped Collection

可以用命令 `convertToCapped` 将一个非 Capped Collection 转成一个 Capped Collection：

```
db.runCommand({"convertToCapped": "mycoll", size: 100000});
```

`size` 字节参数指定了 Capped Collection 的大小。

注意，这个命令将获得一个全局写锁并将阻塞其他操作，直到它完成为止。

在规定的时间周期之后自动移除数据

当需要设置数据过期时，可以考虑使用 MongoDB 的 TTL 索引。

TTL Collections 与 Capped Collection 不兼容。



Tailable Cursor (结尾光标)

可以在 Capped Collection 中使用 Tailable Cursor。同 UNIX 的 `tail -f` 命令相似, Tailable Cursor 用来获取一个 Capped Collection 的结尾内容。随着新文档被插入 Capped Collection, 我们能使用 Tailable Cursor 继续检索文档。

如果想了解如何创建一个 Tailable Cursor, 则可以参阅 <https://docs.mongodb.com/manual/core/tailable-cursors/>。

3. Document (文档)

MongoDB 将数据的记录作为 BSON 文档进行存储。BSON 是 JSON 文档的二进制表示, 但拥有比 JSON 更多的数据类型。如果想了解 BSON 规范的相关内容, 则可以参阅 <http://bsonspec.org/>, 或者 <https://docs.mongodb.com/manual/reference/bson-types/>。

文档的结构

MongoDB 的文档由 field value (字段/值) 对组成, 如下所示:

```
{
  field1: value1,
  field2: value2,
  field3: value3,
  ...
  fieldN: valueN
}
```

字段的值可以是任意 BSON 数据类型, 包括其他文档、数组及文档数组。例如, 下面的文档包含不同类型的值:

```
var mydoc = {
  _id: ObjectId("5099803df3f4948bd2f98391"),
  name: { first: "Alan", last: "Turing" },
  birth: new Date('Jun 23, 1912'),
  death: new Date('Jun 07, 1954'),
  contribs: [ "Turing machine", "Turing test", "Turingery" ],
  views : NumberLong(1250000)
}
```

上面字段中分别包括以下数据类型:

- `_id` 是一个 `ObjectId`;
- `name` 是一个嵌入式的文档, 包含字段的 `first` 和 `last`;



- birth 和 death 保存的是 Date 类型的值；
- contribs 保存的是 string 的 array；
- views 保存的是 NumberLong 类型的值。

字段名称

字段名称是字符串。文档中对于字段名称有如下限制：

- 字段名称_id 被保留用于作为主键，其值必须是集合中唯一的，是不可变的，并且可以是除 array 外的任何类型；
- 该字段名称不能以美元符号“\$”字符开头；
- 字段名称不能包含点“.”字符；
- 字段名称不能包含空（null）字符。

BSON 文档有多个字段可能具有相同名称。大多数 MongoDB 接口被用来代表一个 MongoDB 结构（例如 hash table），不支持重复的字段名称。如果需要操纵包含具有相同名称的多个字段的文档，请参阅 MongoDB 驱动程序的相关内容，见 <https://docs.mongodb.com/manual/applications/drivers/>。

内部 MongoDB 进程创建的一些文件可能有重复的字段，但是 MongoDB 进程不会不断将重复字段增加到现有用户的文档中。

字段值限制

索引的集合，其值受到字段值 Maximum Index Key Length 的限制。详情可以参阅 <https://docs.mongodb.com/manual/reference/limits/#Index-Key-Limit>。

5.7.3 MongoDB 的数据模型

1. 数据建模简介

MongoDB 中的数据有一个灵活的模式。不像 SQL 数据库，必须要在插入数据之前声明一个表的确定模式。MongoDB 的集合并不强制文档的结构，以便其能灵活便利地将文件映射到一个实体或对象。每个文档可以匹配所要表示实体的数据字段，即使数据的变化很显著。但在实际操作中，一个集合的文档共享一个相似的结构。

数据模型的关键挑战在于平衡应用的需要、数据库引擎的性能和数据存取模式。当设计数据模型时，要考虑数据在应用里的使用情况（例如查询、更新和处理数据），以及数据本身的内在结构。



文档结构

为 MongoDB 应用设计数据模型时的关键是围绕文档的结构和应用如何表示数据间的联系。有两个工具来允许应用表示这些关系：引用和嵌入文档。

引用 (References)

引用通过文档之间的链接 link 或引用 (references) 存储数据间的关系。应用能够解析这些引用来访问到相关数据。从广义上说，这些都是 normalized data model (规范化的数据模型)。

MongoDB 引用如图 5-31 所示。

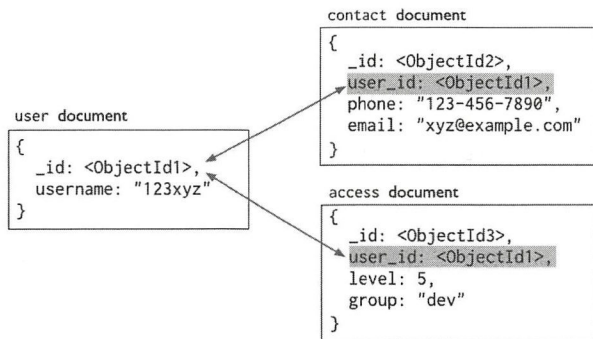


图 5-31 MongoDB 引用

图 5-31 中的数据模型使用引用来连接文档。contract 文档和 access 文档都保存着 user 文档的引用。

下面介绍 normalized data model 在使用引用时的优缺点。

normalized data model 使用引用来描述文档间的关系。一般使用 normalized data model 的情况有：

- 当嵌入会导致数据重复且不具备有效的读性能优势时；
- 表示更复杂的多对多的关系时；
- 当对大型分级数据建模引用比嵌入式文档的灵活性更大时。

同时需要注意，由于使用 normalized data model 需要更多地往返服务器，客户端应用必须处理引用带来的查询问题。

嵌入式数据 (Embedded Data)

嵌入式文档通过在一个单一文档结构里存储相关数据来捕获数据间的关系。MongoDB 的文档使在一个文档里的一个字段或字段数据嵌入一个文档作为子文档具备了可能性。这些非规范化数据使得应用可以在一个单一数据库操作里获取和操纵数据。



MongoDB 嵌入如图 5-32 所示。图 5-32 的数据模型用嵌入式字段来存储所有的相关信息。



图 5-32 MongoDB 嵌入

下面讨论嵌入子文档的数据模型的优缺点。

使用 MongoDB，我们可以在一个单一结构或文档中嵌入相关数据。这个模型是著名的“非规范化”模型，利用了 MongoDB 丰富文档的优势。

嵌入数据模型允许应用在相同的数据库记录里存储相关片段信息。因此，应用在完成一个常规操作时，只需处理很少的查询或更新。

一般来说，当遇到这些情形时可使用嵌入数据模型：实体间有“包含关系”或实体间有一对多的关系。在这些关系里，“多”或子文档经常被看作“一”或父文档的上下文。一般来说，嵌入提供了更好的读性能，以及在单一数据库操作里请求和获取相关数据的能力。嵌入数据模型使得在哪一个原子操作里更新相关数据都成为可能。

然而，在一个文档中嵌入数据模型可能导致文档创建后的增长。文档的增长会影响写性能并产生数据碎片问题。并且，在 MongoDB 里的文档大小必须小于最大的 BSON 文档大小。对大型二进制数据，考虑使用 GridFS。

写操作的原子性

在 MongoDB 中，写操作在文档这一级是原子的，并且没有单一的写操作能原子性地影响多个文档或集合。一个有嵌入数据的非规范化数据模型在一个单一文档里包含能表示一个实体的相关数据。这有利于写操作的原子性，因为单一的写操作能直接对一个实体插入或更新数据。规范化数据在多个集合里分散了数据，这会要求多次写操作，因此不是原子性的。

然而，有利于原子性写的模式会限制一个应用使用数据的方法或修改数据的方法。因此需要平衡原子性和平衡性。

文档增长

有的更新，比如向数组添加元素或添加新的字段，会增加文档的大小。如果文档的大小超过了给该文档分配的空间，MongoDB 就会重新定位这个文档。文档的增长会影响对规范化和非规范化数据的选择。

数据使用 and 性能

在设计一个文档模型时，要考虑应用将如何使用数据库。如果应用仅使用最近插入的数据，则考虑使用 Capped Collection；如果应用总是需要读操作，则添加索引是常见的提升性能的办法。

2. 数据模型的例子和模式

下面介绍 MongoDB 各种数据建模的模式和公共模式设计。

嵌入式文档的一对一关系

考虑这么一个示例，一个 address（地址）属于一个 patron（顾客），它们是一对一的关系。

在 normalized data model 里面，address 文档包含 patron 文档的引用。

```
{
  _id: "joe",
  name: "Joe Bookreader"
}

{
  patron_id: "joe",
  street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345"
}
```

如果 address 数据被频繁使用 name 信息检索，然后找到引用，则应用程序需要发出多个查询来解析引用。因此，更好的数据模型应该是将 address 数据嵌入 patron 数据中，如下面的文档：

```
{
  _id: "joe",
  name: "Joe Bookreader",
  address: {
    street: "123 Fake Street",
    city: "Faketon",
```

```
        state: "MA",
        zip: "12345"
    }
}
```

这样，应用程序只需要一次查询，就能获取完整的 patron 信息。

嵌入式文档的一对多关系

如果一个 patron（顾客）拥有多个 address（地址），则它们是一对多的关系。

在 normalized data model 里面，几个 address 文档都包含 patron 文档的引用。

```
{
  _id: "joe",
  name: "Joe Bookreader"
}

{
  patron_id: "joe",
  street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345"
}

{
  patron_id: "joe",
  street: "1 Some Other Street",
  city: "Boston",
  state: "MA",
  zip: "12345"
}
```

如果 address 数据被频繁地使用 name 信息检索，然后找到引用，则应用程序需要发出多个查询来解析引用。因此，更好的数据模型应该是将 address 数据嵌入 patron 数据中，如下面的文档：

```
{
  _id: "joe",
  name: "Joe Bookreader",
  addresses: [
```



```

        {
            street: "123 Fake Street",
            city: "Faketon",
            state: "MA",
            zip: "12345"
        },
        {
            street: "1 Some Other Street",
            city: "Boston",
            state: "MA",
            zip: "12345"
        }
    ]
}

```

这样，应用程序只需要一次查询，就能获取完整的 patron 信息。

引用文档的一对多关系

考虑这么一个示例，publisher（出版商）与 book（书）的关系。

嵌入 publisher 文档到 book 文档里面，会导致 publisher 数据的重复，如下面的文档：

```

{
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English",
    publisher: {
        name: "O'Reilly Media",
        founded: 1980,
        location: "CA"
    }
}

{
    title: "50 Tips and Tricks for MongoDB Developer",
    author: "Kristina Chodorow",
    published_date: ISODate("2011-05-06"),

```

```
pages: 68,  
language: "English",  
publisher: {  
  name: "O'Reilly Media",  
  founded: 1980,  
  location: "CA"  
}
```

为了避免 publisher 数据的重复，使用引用来保持 publisher 与 book 的集合是独立的集合。如下面的例子：

```
{  
  name: "O'Reilly Media",  
  founded: 1980,  
  location: "CA",  
  books: [123456789, 234567890, ...]  
}  
  
{  
  _id: 123456789,  
  title: "MongoDB: The Definitive Guide",  
  author: [ "Kristina Chodorow", "Mike Dirolf" ],  
  published_date: ISODate("2010-09-24"),  
  pages: 216,  
  language: "English"  
}  
  
{  
  _id: 234567890,  
  title: "50 Tips and Tricks for MongoDB Developer",  
  author: "Kristina Chodorow",  
  published_date: ISODate("2011-05-06"),  
  pages: 68,  
  language: "English"  
}
```

为了避免可变的、不断增长的数组，可以考虑将 publisher 的引用保存在 book 的文档里，如下面的例子：

```
{
  _id: "oreilly",
  name: "O'Reilly Media",
  founded: 1980,
  location: "CA"
}

{
  _id: 123456789,
  title: "MongoDB: The Definitive Guide",
  author: [ "Kristina Chodorow", "Mike Dirolf" ],
  published_date: ISODate("2010-09-24"),
  pages: 216,
  language: "English",
  publisher_id: "oreilly"
}

{
  _id: 234567890,
  title: "50 Tips and Tricks for MongoDB Developer",
  author: "Kristina Chodorow",
  published_date: ISODate("2011-05-06"),
  pages: 68,
  language: "English",
  publisher_id: "oreilly"
}
```

父引用的树结构

Parent Reference（父引用）模式将每个树节点存储在文档中；除了树节点，文档还存储节点的父节点的id。

考虑下面“类别”的层次结构。“类别”的树形结构如图 5-33 所示。

下面使用 Parent Reference 将父“类别”存储到字段 parent 里面：

```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "dbm", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", parent: "Books" } )
db.categories.insert( { _id: "Books", parent: null } )
```

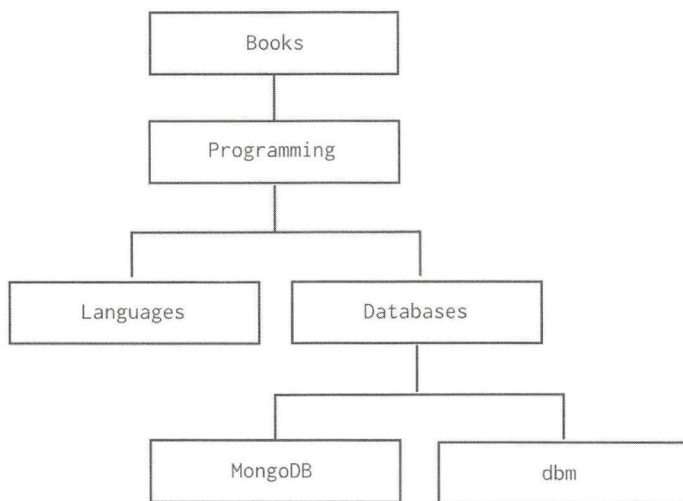


图 5-33 “类别”的树形结构

检索父节点的查询是快速和简单的：

```
db.categories.findOne( { _id: "MongoDB" } ).parent
```

可以在字段 `parent` 上创建索引，这样就能通过父节点快速检索：

```
db.categories.createIndex( { parent: 1 } )
```

可以通过 `parent` 字段查询找到它的直接子节点：

```
db.categories.find( { parent: "Databases" } )
```

Parent Link 模式提供了简单的树存储解决方案，但需要多次查询检索子树。

子引用的树结构

Child Reference（子引用）模式将每个树节点存储在文档中；除了树节点，文档还存储了节点的子节点的 `id` 数组。

考虑图 5-33 的层次结构。下面使用 Child Reference 来存储子节点到字段 `children` 里面：

```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )
db.categories.insert( { _id: "Programming", children: [ "Databases",
"Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```


检索子节点的查询是快速和简单的：

```
db.categories.findOne( { _id: "Databases" } ).children
```

可以在字段 `children` 上创建索引，这样就能通过子节点来快速检索：

```
db.categories.createIndex( { children: 1 } )
```

可以通过 `children` 字段查询找到它的父节点及兄弟节点：

```
db.categories.find( { children: "MongoDB" } )
```

Child Reference 模式提供了一个不对子树进行必要操作的树存储解决方案。该模式同样适用于存储多个父节点的图结构。

Array of Ancestors 的树结构

Array of Ancestors（祖先数组）模式存储每个树节点在文档中；除了树节点，文档还存储了节点的祖先的 ID 或路径数组。

考虑图 5-33 的层次结构。下面是使用 Array of Ancestors 的例子，除了 `ancestors` 字段，还存储父“类别”到字段 `parent` 里面：

```
db.categories.insert( { _id: "MongoDB", ancestors: [ "Books", "Programming",  
"Databases" ], parent: "Databases" } )  
db.categories.insert( { _id: "dbm", ancestors: [ "Books", "Programming",  
"Databases" ], parent: "Databases" } )  
db.categories.insert( { _id: "Databases", ancestors: [ "Books",  
"Programming" ], parent: "Programming" } )  
db.categories.insert( { _id: "Languages", ancestors: [ "Books",  
"Programming" ], parent: "Programming" } )  
db.categories.insert( { _id: "Programming", ancestors: [ "Books" ], parent:  
"Books" } )  
db.categories.insert( { _id: "Books", ancestors: [ ], parent: null } )
```

检索节点的祖先或者路径的查询是快速和简单的：

```
db.categories.findOne( { _id: "MongoDB" } ).ancestors
```

可以在字段 `ancestors` 上创建索引，这样就能通过祖先节点来进行快速检索：

```
db.categories.createIndex( { ancestors: 1 } )
```

可以通过 `ancestors` 字段查询找到它所有的后代：

```
db.categories.find( { ancestors: "Programming" } )
```

Array of Ancestors 模式提供了一种解决方案，通过在 `ancestors` 字段创建索引，从而可以快

速、高效地查找一个节点的后代和祖先。Array of Ancestors 很适合处理子树相关的工作。

Array of Ancestors 模式比 Materialized Paths 模式稍微慢一点，但更容易使用。

Materialized Paths 的树结构

Materialized Paths 模式存储每个树节点在文档中；除了树节点，文档还存储了节点的祖先的 ID 字符串或路径。虽然 Materialized Paths 模式需要额外的字符串和正则表达式的工作，但该模式具有更加灵活的路径处理，比如通过局部路线来查找寻找节点。

考虑图 5-33 的层次结构。下面是使用 Materialized Paths 的例子，将路径存储到字段 path，其中路径字符串使用逗号作为分隔符：

```
db.categories.insert( { _id: "Books", path: null } )
db.categories.insert( { _id: "Programming", path: ",Books," } )
db.categories.insert( { _id: "Databases", path: ",Books,Programming," } )
db.categories.insert( { _id: "Languages", path: ",Books,Programming," } )
db.categories.insert( { _id: "MongoDB", path: ",Books,Programming,Databases," } )
db.categories.insert( { _id: "dbm", path: ",Books,Programming,Databases," } )
```

可以检索整个树，通过 path 字段来排序：

```
db.categories.find().sort( { path: 1 } )
```

可以在字段 path 上使用正则表达式来查询 Programming 后代：

```
db.categories.find( { path: /,Programming,/ } )
```

当 Books 是最高层时，也可以使用下面的方式来检索 Books 的后代：

```
db.categories.find( { path: /^,Books,/ } )
```

可以在字段 path 上创建索引：

```
db.categories.createIndex( { path: 1 } )
```

索引的查询性能依赖于以下的查询方式：

- 如果从根 Books 子树（例如，/,Books,/ 或 /^,Books,Programming,/）进行查询，则在 path 字段上建立索引可以显著提高查询性能。
- 若查询里面没有提供根的路径（例如，/,Databases,/），或要查询的子树节点在索引字符串的中间位置，则查询必须检查整个索引。对于这类查询，当索引比整个集合小很多时，可以提升一些性能。

Nested Sets（嵌套集合）的树结构

Nested Sets 模式定义每个树中的节点作为树往返的点。应用程序访问两次树中的每个节点，第一次是去程，第二次是返程。Nested Sets 模式存储每个树节点在文档中；除了树节点，文档

还存储了节点的父亲（节点的去程点）在 `left` 字段，存储其返程点在 `right` 字段。

Nested Sets 模式结构如图 5-34 所示。

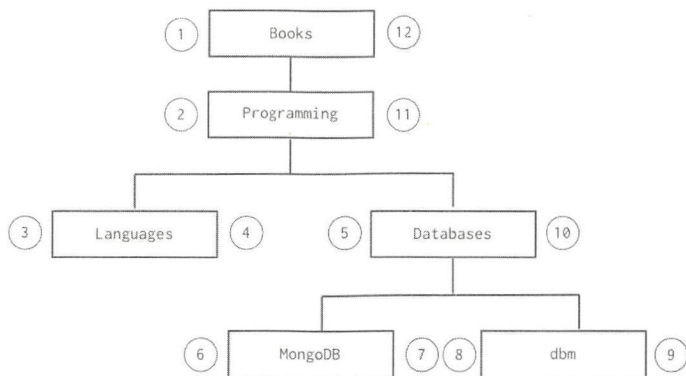


图 5-34 Nested Sets 模式结构

考虑图 5-34 的层次结构。下面是使用 Nested Sets 的例子：

```

db.categories.insert( { _id: "Books", parent: 0, left: 1, right: 12 } )
db.categories.insert( { _id: "Programming", parent: "Books", left: 2, right: 11 } )
db.categories.insert( { _id: "Languages", parent: "Programming", left: 3, right: 4 } )
db.categories.insert( { _id: "Databases", parent: "Programming", left: 5, right:
10 } )
db.categories.insert( { _id: "MongoDB", parent: "Databases", left: 6, right: 7 } )
db.categories.insert( { _id: "dbm", parent: "Databases", left: 8, right: 9 } )
  
```

可以检索节点的后代：

```

var databaseCategory = db.categories.findOne( { _id: "Databases" } );
db.categories.find( { left: { $gt: databaseCategory.left }, right: { $lt:
databaseCategory.right } } );
  
```

Nested Sets 模式提供用于查找子树快速而有效的解决方案，对于修改的树结构则比较低效。因此，这种模式最适合不会改变的静态树。

原子操作的模型数据

在 MongoDB 中，写操作如 `db.collection.update()`、`db.collection.findAndModify()`、`db.collection.remove()`等在单文档级别是原子的。字段必须一起更新，以确保嵌入在同一文件中的字段可以用原子方式更新。

例如，以图书管理为例，想保存图书信息，包括可用于出售的数量（`available`），以及当前的售出信息的情况（`checkout`）。这本书可用于出售的数量和售出的信息应该是同步的。因此，

在同一个文档中嵌入 available 字段和 checkout 字段，确保可以原子更新两个字段。

```
{
  _id: 123456789,
  title: "MongoDB: The Definitive Guide",
  author: [ "Kristina Chodorow", "Mike Dirolf" ],
  published_date: ISODate("2010-09-24"),
  pages: 216,
  language: "English",
  publisher_id: "oreilly",
  available: 3,
  checkout: [ { by: "joe", date: ISODate("2012-10-15") } ]
}
```

在更新售出信息时，只要使用 db.collection.update() 方法，就能同时更新 available 字段和 checkout 字段。

```
db.books.update (
  { _id: 123456789, available: { $gt: 0 } },
  {
    $inc: { available: -1 },
    $push: { checkout: { by: "abc", date: new Date() } }
  }
)
```

该操作会返回 WriteResult() 对象，包含操作的状态信息。

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

nMatched 字段显示有 1 个文档匹配更新的条件，nModified 字段显示更新了 1 个文档。

如果没有文档匹配上更新条件，则 nMatched 和 nModified 都是 0，说明不能售出书。

支持关键字搜索的模型数据

可以在字符串数组上创建多键索引（multi-key index）。比如下面的例子：

```
{ title : "Moby-Dick" ,
  author : "Herman Melville" ,
  published : 1851 ,
  ISBN : 0451526996 ,
  topics : [ "whaling" , "allegory" , "revenge" , "American" ,
    "novel" , "nautical" , "voyage" , "Cape Cod" ]
}
```


可以在 topics 数组上创建多键索引。

```
db.volumes.createIndex( { topics: 1 } )
```

多键索引为 topics 中的每个关键字都创建了单独的索引条目。例如，索引包含条目 whaling 及另外一个条目 allegory。

然后，可以执行基于关键字的查询，例如：

```
db.volumes.findOne( { topics : "voyage" }, { title: 1 } )
```

注意，如果在数组中有大量的元素，有几百或几千个关键字，则都将导致更大的插入索引的成本。

MongoDB 支持使用特定的数据模型和多键索引关键字来进行搜索。然而，这些关键字索引在下面几个方面并不非常有效或合适：

- Stemming（词干）——MongoDB 中关键字查询无法解析根或相关词的关键字。
- Synonyms（同义词）——基于关键字的搜索功能，必须在应用层提供同义词或相关查询。
- Ranking（排行）——本文档中描述的关键字的查询窗口，不提供对结果的加权。
- Asynchronous Indexing（异步索引）——MongoDB 构建索引是同步的，这意味着用于关键字的索引将更加实时。然而，异步索引可能对某些类型的内容和工作负载更有效。

Monetary Data（货币数据）模型

在 MongoDB 中存储数字数据支持 IEEE 754 标准的 64 位浮点数，或者 32 位/64 位有符号整数，以满足处理货币数据的需求。

下面介绍在 MongoDB 中对货币数据进行建模的两种方法。

（1）精确精密（Exact Precision）模型。

要使用 Exact Precision 进行建模：

- 首先，确定所需的货币价值的最大精度。例如，你的应用程序可能需要精确到 1 分钱的 1/10；
- 其次，通过 10 的幂运算，将货币价值转为整数。例如，如果所需要的最大精度为 1 分钱的 1/10，则货币价值需乘以 1000。
- 最后，存储转换货币价值。

例如，下面将 9.99 美元通过 1000 的比例来保持精确到 1 分钱的 1/10。

```
{ price: 9990, currency: "USD" }
```

（2）任意精确（Arbitrary Precision）模型。

任意精度模型使用的值为两个字段：

- 第一个字段，将该货币值编码为非数值数据类型，例如 `BinData` 或 `string`；
- 第二个字段，存储该值的双精度浮点近似值。

下面的示例使用任意精度模型来存储 9.99 美元的价格和 0.25 美元的费用：

```
{
  price: { display: "9.99", approx: 9.9900000000000002, currency: "USD" },
  fee: { display: "0.25", approx: 0.2499999999999999, currency: "USD" }
}
```

Time Data（时间数据）模型

MongoDB 默认存储 UTC 时间类型，会将本地时间转为该类型来展示。在必须对一些未经修改的本地时间值进行操作或报告时，应用程序可以在 UTC 时间戳旁存储时区，并在其应用逻辑里计算原有的本地时间。

例如，在 MongoDB shell 中可以同时存储当前时间，以及当前客户端的 UTC 的偏移量：

```
var now = new Date();
db.data.save( { date: now,
                 offset: now.getTimezoneOffset() } );
```

可以通过应用保存的偏移量重建原始本地时间：

```
var record = db.data.findOne();
var localNow = new Date( record.date.getTime() - ( record.offset * 60000 ) );
```

5.7.4 示例：Java 连接 MongoDB

下面介绍用 Java 语言连接 MongoDB 数据库的步骤。其他编程语言的驱动的支持情况可以在线参阅：<https://docs.mongodb.com/ecosystem/drivers/>。

（1）下载和安装 MongoDB Java Driver 和 BSON 库。

下载和安装可以参阅 <http://mongodb.github.io/mongo-java-driver/>。

（2）连接 MongoDB。

使用 `com.mongodb.MongoClient` 类连接运行中的 `mongodb` 实例。使用 `com.mongodb.client.MongoDatabas` 接口访问 MongoDB 中特定的数据库。

导入包：

```
import com.mongodb.MongoClient;
```

```
import com.mongodb.client.MongoDatabase;
```

连接运行在 localhost（默认端口是 27017）上的 mongodb 实例：

```
MongoClient mongoClient = new MongoClient();
```

当然，也可以指定 IP 和端口号：

```
MongoClient mongoClient2 = new MongoClient("localhost", 27017);
```

在连接成功后，就可以访问 test 数据库了：

```
MongoDatabase db = mongoClient.getDatabase("test");
```

5.8 实战：基于 Redis 的分布式锁

在 1.8 节中，我们介绍了多线程在并发时可能产生的线程安全问题。为了避免该类安全问题，通常会使用同步或原子访问来实现线程间的资源访问的限制。但是这种方式只适合于单机部署的场景。那么，如何解决在分布式环境下的并发问题呢？可以使用分布式锁来实现分布式下资源的安全访问。

下面我们将介绍如何基于 Redis 来实现分布式下的加锁和解锁功能。

5.8.1 项目概述

我们将创建一个名为“redis-lock”的应用。在该应用中，我们演示基于 Redis 来实现分布式下的加锁和解锁功能。

为了能够正常运行该应用，需要在应用中添加如下依赖：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.2.RELEASE</version>
</parent>

<dependencies>
    <dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>2.9.0</version>
    </dependency>
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
</dependencies>
```

其中，我们采用 Redis 的 Java 客户端 Jedis，同时使用 Spring Boot 作为应用的快速启动和测试框架。有关 Spring Boot 的内容，会在 8.2 节进行详细介绍。

5.8.2 项目配置

在“com.waylau.redis.config”包下创建 Redis 的配置类。为了方便测试，我们只在本地使用一个 Redis 节点。

```
package com.waylau.redis.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;

@Configuration
public class JedisConfig {

    private static JedisPool jedisPool;

    @Bean
    public Jedis getBuild() {
        jedisPool = new JedisPool("localhost", 6379);
        Jedis jedis = jedisPool.getResource();
```



```

        return jedis;
    }
}

```

5.8.3 编码实现

定义如下分布式锁的接口：

```

public interface IRedisLock {

    Boolean lock(int lockKeyExpireSecond, String lockName, Boolean isWait)
throws Exception;

    Boolean unlock(String lockName) throws Exception;
}

```

分布式锁的实现如下：

```

package com.waylau.redis.lock;

import java.util.UUID;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;

import redis.clients.jedis.Jedis;

@Component
public class RedisLock implements IRedisLock {

    private static org.slf4j.Logger logger = org.slf4j.LoggerFactory.getLogger
(RedisLock.class);

    /**
     * 当前机器节点锁标识
     */
    private static String redisIdentityKey = UUID.randomUUID().toString();
}

```

```
/**
 * 获取当前机器节点在锁中的标识符
 */
public static String getRedisIdentityKey() {
    return redisIdentityKey;
}

/**
 * 等待获取锁的时间，可以根据当前任务的执行时间来设置
 */
private static final long WaitLockTimeSecond = 2000;

/**
 * 重试获取锁的次数，可以根据当前任务的执行时间来设置
 * 需要时间=RetryCount*(WaitLockTimeSecond/1000)
 */
private static final int RetryCount = 10;

@Autowired
private Jedis jedis;

@Override
public Boolean lock(int lockNameExpireSecond, String lockName, Boolean
isWait) throws Exception {
    if (StringUtils.isEmpty(lockName))
        throw new Exception("lockName is empty.");

    int retryCounts = 0;
    while (true) {
        Long status, expire = 0L;
        status = jedis.setnx(lockName, redisIdentityKey);
        if (status > 0) {
            expire = jedis.expire(lockName, lockNameExpireSecond);
        }
        if (status > 0 && expire > 0) {
            logger.info(String.format("t:%s, 当前节点: %s, 获取锁: %s",
                Thread.currentThread().getId(),
                getRedisIdentityKey(),
```



```
        lockName));  
        return true;/** 获取到 lock */  
    }  
  
    try {  
        if (isWait && retryCounts < RetryCount) {  
            retryCounts++;  
            synchronized (this) {  
                logger.info(String.format("t:%s,当前节点: %s,尝试等待获  
取锁: %s",  
                    Thread.currentThread().getId(),  
                    getRedisIdentityKey(),  
                    lockName));  
  
                // 未能获取锁,进行指定时间的 wait 后再重试  
                this.wait(WaitLockTimeSecond);  
            }  
        } else if (retryCounts == RetryCount) {  
            logger.info(String.format("t:%s,当前节点: %s,指定时间内获取  
锁失败: %s",  
                Thread.currentThread().getId(),  
                getRedisIdentityKey(),  
                lockName));  
            return false;  
        } else {  
            return false;  
        }  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}  
  
@Override  
public Boolean unlock(String lockName) throws Exception {  
    if (StringUtils.isEmpty(lockName))  
        throw new Exception("lockName is empty.");
```



```

        long status = jedis.del(lockName);
        if (status > 0) {
            logger.info(String.format("t:%s,当前节点: %s,释放锁: %s 成功。",
                Thread.currentThread().getId(),
                getRedisIdentityKey(),
                lockName));
            return true;
        }
        logger.info(String.format("t:%s,当前节点: %s,释放锁: %s 失败。",
            Thread.currentThread().getId(),
            getRedisIdentityKey(),
            lockName));
        return false;
    }
}

```

5.8.4 运行

为了方便测试，编写如下测试用例：

```

package com.waylau.redis;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import com.waylau.redis.lock.RedisLock;

@RunWith(SpringRunner.class)
@SpringBootTest
public class ApplicationTests {

    @Autowired
    private RedisLock redisLock;
}

```




```
@Test
public void contextLoads() throws Exception {
    String lockName = "waylau.com";
    int lockNameExpireSecond = 10;

    for (int i = 0; i < 4; i++) {
        redisLock.lock(lockNameExpireSecond, lockName, true);

        Thread.sleep(3000);

        redisLock.unlock(lockName);
    }
}
```

先启动 Redis 服务，再执行该测试用例。我们分别启动两个测试用例，以模拟两个客户端同时竞争锁的场景。在程序执行后，观察控制台的输出信息。

第一个测试用例的输出如下：

```
2018-05-16 00:19:42.286 INFO 14904 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1,当前节点:
a76ecccb-e5a7-4680-934f-a2b41b07e06f,获取到锁: waylau.com
2018-05-16 00:19:45.286 INFO 14904 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1,当前节点:
a76ecccb-e5a7-4680-934f-a2b41b07e06f,释放锁: waylau.com 成功。
2018-05-16 00:19:45.287 INFO 14904 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1,当前节点:
a76ecccb-e5a7-4680-934f-a2b41b07e06f,获取到锁: waylau.com
2018-05-16 00:19:48.288 INFO 14904 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1,当前节点:
a76ecccb-e5a7-4680-934f-a2b41b07e06f,释放锁: waylau.com 成功。
2018-05-16 00:19:48.290 INFO 14904 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1,当前节点:
a76ecccb-e5a7-4680-934f-a2b41b07e06f,获取到锁: waylau.com
2018-05-16 00:19:51.290 INFO 14904 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1,当前节点:
a76ecccb-e5a7-4680-934f-a2b41b07e06f,释放锁: waylau.com 成功。
2018-05-16 00:19:51.291 INFO 14904 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1,当前节点:
```



```
a76eccc-b-e5a7-4680-934f-a2b41b07e06f, 获取到锁: waylau.com
2018-05-16 00:19:54.293 INFO 14904 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1, 当前节点:
a76eccc-b-e5a7-4680-934f-a2b41b07e06f, 释放锁: waylau.com 成功。
```

第二个测试用例的输出如下:

```
2018-05-16 00:19:45.118 INFO 15328 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1, 当前节点:
d7717168-1471-4590-b938-9eb58bd88b59, 尝试等待获取锁: waylau.com
2018-05-16 00:19:47.119 INFO 15328 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1, 当前节点:
d7717168-1471-4590-b938-9eb58bd88b59, 尝试等待获取锁: waylau.com
2018-05-16 00:19:49.121 INFO 15328 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1, 当前节点:
d7717168-1471-4590-b938-9eb58bd88b59, 尝试等待获取锁: waylau.com
2018-05-16 00:19:51.122 INFO 15328 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1, 当前节点:
d7717168-1471-4590-b938-9eb58bd88b59, 尝试等待获取锁: waylau.com
2018-05-16 00:19:53.123 INFO 15328 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1, 当前节点:
d7717168-1471-4590-b938-9eb58bd88b59, 尝试等待获取锁: waylau.com
2018-05-16 00:19:55.125 INFO 15328 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1, 当前节点:
d7717168-1471-4590-b938-9eb58bd88b59, 获取到锁: waylau.com
2018-05-16 00:19:58.126 INFO 15328 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1, 当前节点:
d7717168-1471-4590-b938-9eb58bd88b59, 释放锁: waylau.com 成功。
2018-05-16 00:19:58.128 INFO 15328 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1, 当前节点:
d7717168-1471-4590-b938-9eb58bd88b59, 获取到锁: waylau.com
2018-05-16 00:20:01.129 INFO 15328 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1, 当前节点:
d7717168-1471-4590-b938-9eb58bd88b59, 释放锁: waylau.com 成功。
2018-05-16 00:20:01.130 INFO 15328 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1, 当前节点:
d7717168-1471-4590-b938-9eb58bd88b59, 获取到锁: waylau.com
2018-05-16 00:20:04.131 INFO 15328 --- [          main]
com.waylau.redis.lock.RedisLock      : t:1, 当前节点:
```



d7717168-1471-4590-b938-9eb58bd88b59, 释放锁: waylau.com 成功。

```
2018-05-16 00:20:04.132 INFO 15328 --- [          main]
```

```
com.waylau.redis.lock.RedisLock          : t:1, 当前节点:
```

d7717168-1471-4590-b938-9eb58bd88b59, 获取到锁: waylau.com

```
2018-05-16 00:20:07.132 INFO 15328 --- [          main]
```

```
com.waylau.redis.lock.RedisLock          : t:1, 当前节点:
```

d7717168-1471-4590-b938-9eb58bd88b59, 释放锁: waylau.com 成功。

从上面例子运行的结果可以看出, 第一个测试用例先运行, 于是先获取到锁(加锁)。第二个测试用例后运行, 在获取锁时先做等待, 直到第一个测试用例解锁后, 方能获取锁。

上述代码可以在 <https://github.com/waylau/distributed-systems-technologies-and-cases-analysis> 的 redis-lock 程序中找到。



第 6 章

分布式监控





6.1 分布式监控概述

相比于单机的部署模式，应用在分布式部署下出现了新的挑战。一方面，应用往往有多个实例，这些实例需要在分布式的多个节点上进行部署；另一方面，通过人工手动操作命令来监控这些应用已经变得越来越困难。特别是在微服务架构下，每个微服务往往都需要设置单独的监控，这意味着越多的服务需要越多的监控。而且每个微服务可能使用不同的技术或语言，依靠不同的机器或容器，使用特有的版本控制，这也大大增加了监控的复杂性。

本章将介绍分布式下常用的监控技术，这些技术能够解决上面提到的问题。

6.1.1 使用场景

如果你的项目是单机部署的应用，或者节点数不超过 3 个，那么即便不使用专业的监控产品，也能满足平常的运维需要。可以手工操作登录部署应用的主机，通过命令行来观察主机资源占用的情况，比如 CPU、硬盘、内存等。也可以通过观察应用的日志文件来排查应用运行过程中的问题。

但是，若部署的节点数达到一定数量，比如 10 个甚至更多，通过手工操作来监测主机便变成了一个不可能完成的任务。一方面，手工操作费时费力，重复性操作令人乏味；另一方面，也是最为重要的一点，手工操作加大了出错的可能性。所以，把重复性的工作交给计算机来做是明智之举。采用成熟的分布式监控产品能帮助你减少不必要的劳动，省心省力。你要做的只是设置必要的执行脚本或报警阈值，分布式监控产品会自动帮你运维。当在运维过程中监测到异常时，你会收到来自监控系统的告警通知。这样，人主动去轮询排查运维问题，转变为被动接收告警通知，从而大大节省了人力。

6.1.2 常用技术

目前，市面上开源的监控产品比较多，功能也很丰富。比如 Nagios 和 Zabbix 便是两款老牌的工业级监控产品，可以满足大部分场景的需求，包括硬件资源和软件资源的监控。Consul 和 ZooKeeper 则更加专注于服务的管理，比如服务的注册与发现、维护服务的高可用等。

6.2 Nagios

Nagios 是一款开源的免费网络监视工具，致力于打造符合行业标准的 IT 基础架构的监控系





统。Nagios 提供了服务器、网络 and 应用的完整的 IT 监控和报警功能，可以有效监控 Windows、Linux 和 UNIX 的主机状态，以及交换机、路由器、打印机等网络设备。在系统或服务状态异常时可以发出邮件或短信报警，第一时间通知网站运维人员，在状态恢复后发出正常的邮件或短信进行通知。

6.2.1 Nagios 监控

本节介绍 Nagios 如何监控不同的服务器、网络设备和网络服务等。

1. 监控 Windows 主机

Nagios 可以监控 Windows 主机的“私有”服务和属性，比如：

- 内存占用率；
- CPU 负荷；
- 硬盘使用率；
- 服务状态；
- 运行的进程；
- 其他。

概览

对 Windows 主机监控私有服务或属性需要在机器上安装 agent 程序。Agent 程序将在监控插件与 Nagios 服务之间起网关代理的作用。如果没有在机器上安装 agent 程序，则 Nagios 将无法对 Windows 私有服务或属性等进行监控。

在下面的例子中，将在 Windows 主机上安装 NSClient++ 构件，并使用 check_nt 插件与 NSClient++ 构件进行通信。

其他的 agent 程序（比如 NC_Net），安装和使用的流程大致相同，可以参考本例。监控 Windows 主机的示意图如图 6-1 所示。

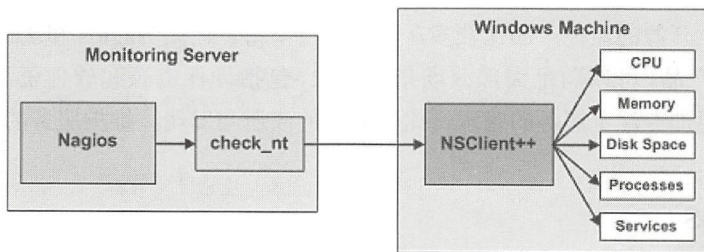


图 6-1 监控 Windows 主机的示意图





步骤

为完成对 Windows 主机的监控，有几个步骤要做：

- 确认一下前提条件；
- 在 Windows 主机上安装 agent 程序（在本例中是安装 NSClient++ 构件）；
- 给 Windows 主机创建新的主机和服务对象定义；
- 重新启动 Nagios 守护进程。

为使整个过程简单化，先完成少量配置文件的工作：

- 把 `check_nt` 命令加入 `commands.cfg` 文件，就可以直接使用 `check_nt` 插件来监控 Windows 服务；
- 一个 Windows 机器的主机对象模板（命名为 `windows-server`）已经在 `templates.cfg` 文件里创建好了，可以很容易地加入一个新的 Windows 主机对象定义。

常用的配置文件可以在 `/usr/local/nagios/etc/objects/` 目录里找到。如果愿意，则可以对里面的对象进行修改以适应你的要求。但是，如果不熟悉如何配置 Nagios，则最好不要这么做。开始时，可以只是按照下面的指令操作来快速完成监控 Windows 主机。

前提条件

首次监控一台 Windows 主机时需要为 Nagios 做点额外的工作。记住，仅仅是监控第一台 Windows 主机时需要做这些工作。

编辑 Nagios 的主配置文件：

```
vi /usr/local/nagios/etc/nagios.cfg
```

把下面这行最前面的“#”号去掉：

```
#cfg_file=/usr/local/nagios/etc/objects/windows.cfg
```

保存配置文件并退出。

上面的工作是让 Nagios 启用 `/usr/local/nagios/etc/objects/windows.cfg` 这个配置文件里的对象定义。在这个配置文件里可以加一些 Windows 的主机或服务的对象定义。该配置文件里已经包含几个主机、主机组及服务对象定义的样例。对于第一台 Windows 主机，可以只是简单地修改里面已经有的主机与服务对象定义而不要新创建一个。

安装 Windows agent 程序

在用 Nagios 监控 Windows 主机的私有服务之前，需要先在主机上安装 agent 程序。推荐使用 NSClient++ 外部构件，它可以在 <http://sourceforge.net/projects/nscplus> 中找到。

下面是安装一个基本的 NSClient++ 外部构件，同时配置 Nagios 来监控 Windows 主机的步骤。



(1) 从 <http://sourceforge.net/projects/nscplus> 页面下载最新稳定版的 NSClient++ 软件包。

(2) 解压软件包到一个目录下，如 C:\NSClient++。

(3) 打开一个命令行窗口并切换到 C:\NSClient++ 目录下。

(4) 用下面的命令将 NSClient++ 系统服务注册到系统里：

```
nsclient++ /install
```

(5) 用下面的命令安装 NSClient++ 系统托盘程序（“SysTray”是大小写敏感的）：

```
nsclient++ SysTray
```

(6) 打开服务管理器并确认 NSClient++ 服务可以在桌面交互（看一下服务管理器里的“Log On”选项卡），如果没有允许桌面交互，则设置为允许，如图 6-2 所示。

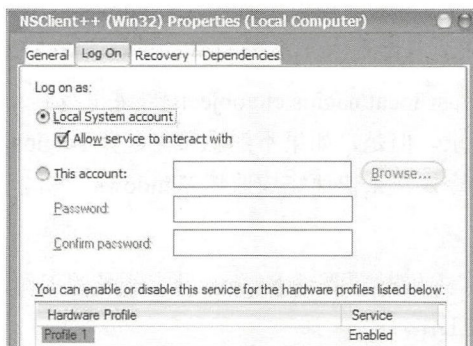


图 6-2 NSClient++ 的设置

(7) 编辑 NSC.INI 文件（位于 C:\NSClient++ 目录）并做如下修改：

- 去掉在 modules 段里的列出模块程序的注释，除了 CheckWMI.dll 和 RemoteConfiguration.dll；
- 最好修改一下在 Settings 段里的“password”选项；
- 去掉 Settings 段里的“allowed_hosts”选项注释，把 Nagios 服务所在主机的 IP 加到这一行里，或置为空，让全部主机都可以联入；
- 确认在 NSClient 段里的“port”选项里已经去掉注释并设置成“12489”（默认端口）。

(8) 用下面的命令启动 NSClient++ 服务：

```
nsclient++ /start
```

(9) 如果安装正确，则一个新的图标会出现在系统托盘里，是一个黄圈里面有个黑色的“M”。

(10) 现在这台 Windows 主机可以加到 Nagios 监控配置里了。



配置 Nagios

为了监控 Windows 主机，下面要在 Nagios 配置文件里添加几个对象定义。

以编辑方式打开 windows.cfg 文件：

```
vi /usr/local/nagios/etc/objects/windows.cfg
```

给 Windows 主机加一个新的主机对象定义以便监控。如果被监控的是第一台 Windows 主机，则可以只修改 windows.cfg 文件里的对象定义。修改 `host_name`、`alias` 和 `address` 字段以符合那台 Windows 主机的实际情况：

```
define host{
    use      windows-server ; Inherit default values from a Windows server
    template (make sure you keep this line!)
    host_name      winserver
    alias          My Windows Server
    address        192.168.1.2
}
```

现在可以在该配置文件里继续添加服务定义，以使 Nagios 监控 Windows 主机上的不同属性内容。如果是第一台 Windows 主机，则可以只修改 windows.cfg 里的服务对象定义。

加入下面的服务定义，用来监控运行在 Windows 主机上的 NSClient++ 外部构件的版本。当要升级 Windows 主机上的外部构件时，这个信息会很有用，因为它可以告知这台 Windows 主机上的 NSClient++ 需要升级到最新版本：

```
define service{
    use      generic-service
    host_name      winserver
    service_description NSClient++ Version
    check_command  check_nt!CLIENTVERSION
}
```

加入下面的服务定义以监控 Windows 主机启动后的运行时间：

```
define service{
    use      generic-service
    host_name      winserver
    service_description Uptime
    check_command  check_nt!UPTIME
}
```

加入下面的服务定义可监控 Windows 主机的 CPU 利用率，并在 5 分钟内 CPU 负荷持续高于 90% 时给出一个紧急警报或高于 80% 时给出一个告警警报：



```
define service{
    use          generic-service
    host_name     winserver
    service_description CPU Load
    check_command check_nt!CPULOAD!-l 5,80,90
}
```

加入下面的服务定义可监控 Windows 主机的内存占用率，并在 5 分钟内内存占用率持续高于 90% 时给出一个紧急警报或高于 80% 时给出一个告警警报：

```
define service{
    use          generic-service
    host_name     winserver
    service_description Memory Usage
    check_command check_nt!MEMUSE!-w 80 -c 90
}
```

加入下面的服务定义可监控 Windows 主机的 C 盘的磁盘利用率，并在磁盘利用率高于 90% 时给出一个紧急警报或高于 80% 时给出一个告警警报：

```
define service{
    use          generic-service
    host_name     winserver
    service_description C:\ Drive Space
    check_command check_nt!USEDISKSPACE!-l c -w 80 -c 90
}
```

加入下面的服务定义可监控 Windows 主机上的 W3SVC 服务状态，并在 W3SVC 服务停止时给出一个紧急警报：

```
define service{
    use          generic-service
    host_name     winserver
    service_description W3SVC
    check_command check_nt!SERVICESTATE!-d SHOWALL -l W3SVC
}
```

加入下面的服务定义可监控 Windows 主机上的 Explorer.exe 进程，并在进程没有运行时给出一个紧急警报：

```
define service{
```



```

use          generic-service
host_name    winserver
service_description Explorer
check_command check_nt!PROCSTATE!-d SHOWALL -l Explorer.exe
}

```

现在已经加好了基础服务定义，保存一下配置文件，就可以监控 Windows 主机了。

口令保护

如果想指定保存在 Windows 主机上 NSClient++ 配置文件里的口令，则可以修改 `check_nt` 命令定义，让它带上口令。以编辑方式打开 `commands.cfg` 文件：

```
vi /usr/local/nagios/etc/objects/commands.cfg
```

修改 `check_nt` 命令的定义，带上“-s”命令参数（这里的 PASSWORD 要换成 Windows 主机的真正口令）：

```

define command{
    command_name    check_nt
    command_line     $USER1$/check_nt -H $HOSTADDRESS$ -p 12489 -s PASSWORD
-v $ARG1$ $ARG2$
}

```

保存文件并退出。

重启 Nagios

如果已经修改好 Nagios 配置文件，则需要先验证你的配置文件，再重启 Nagios。

如果验证配置文件过程中有什么错误信息，则在做下一步前一定要修正好配置文件。一定要保证验证过程中不再有出错信息再启动或重启 Nagios。

2. 监控 Linux/UNIX 主机

Nagios 可以监控 Linux/UNIX 主机的“私有”服务和属性，比如：

- 内存占用率；
- CPU 负荷；
- 硬盘使用率；
- 登录的用户；
- 运行的进程；
- 其他。



对 Linux/UNIX 主机进行监控有多种方式。一种方式是使用共享 SSH 密钥和 `check_by_ssh` 插件来执行对远程主机的监控。这种方法会导致安装 Nagios 的监控服务器承受很高的系统负荷，尤其是要监控成百个主机中的上千个服务时。其主要原因是，建立和销毁 SSH 连接会产生开销。

另一种方式是使用 NRPE 外部构件监控远程主机。NRPE 外部构件可以在远程的 Linux/UNIX 主机上执行插件程序。如果要像监控本地主机一样对远程主机的磁盘利用率、CPU 负荷和内存占用率等情况进行监控，则 NRPE 外部构件非常有用。

监控 Linux/UNIX 主机的示意图如图 6-3 所示。

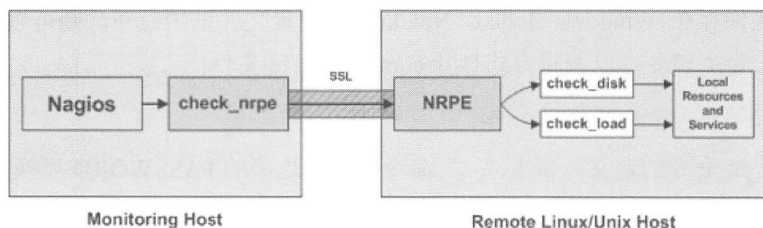


图 6-3 监控 Linux/UNIX 主机的示意图

本书不会对 NRPE 的使用过程做过多的介绍，读者可以自行查阅在线文档 <https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/addons.html#nrpe>。

3. 监控路由器和交换机

Nagios 可以监控路由器和交换机的状态。一些便宜的“无网管”功能的交换机与集线器不能配置 IP 地址，而且对于网络是不可见的组成构件，因而没办法监控这类东西。稍贵一些的交换机和路由器可以配置 IP 地址，可以用 ping 检测或通过 SNMP 来查询状态信息。

下面将描述如何监控这些有网管功能的交换机、集线器和路由器，监控的内容如下：

- 丢包率，RTA（平均回包周期）；
- SNMP 状态信息；
- 带宽与流量。

概览

监控交换机与路由器可简可繁，主要看拥有什么样的设备与想监控什么内容。作为极为重要的网络组成构件，毫无疑问至少要监控一些基本状态。

交换机与路由器可以简单地用 ping 来监控丢包率、RTA 等数据。如果交换机支持 SNMP，则可以用 `check_snmp` 插件来监控端口状态等，也可以用 `check_mrtgtraf` 插件来监控带宽（如果用了 MRTG）。

只有当系统里安装了 `net-snmp` 和 `net-snmp-utils` 包后 `check_snmp` 插件才编译。先确定插件



已经在 `/usr/local/nagios/libexec` 目录里再继续做，如果没有这个文件，则先安装 `net-snmp` 和 `net-snmp-utils` 包并重编译后重新安装 Nagio 插件包。

监控交换机与路由器的示意图如图 6-4 所示。

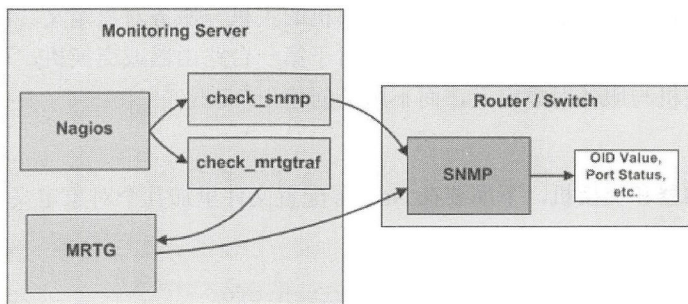


图 6-4 监控交换机与路由器的示意图

步骤

为了完成对交换机与路由器的监控，有几个步骤要做：

- 确认一下前提条件；
- 给设备创建要监控的主机与服务对象定义；
- 重启动 Nagios 守护进程。

为使整个过程简单化，先完成少量配置文件的工作：

- 把 `check_snmp` 和 `check_local_mrtgtraf` 命令加入 `commands.cfg` 文件，可以用 `check_snmp` 和 `check_mrtgtraf` 插件来监控网络路由器；
- 一个交换器的主机对象模板（命名为 `generic-switch`）已经在 `templates.cfg` 文件里创建好了，可以更容易地加入一个新的交换机与路由器的主机对象定义。

常用的配置文件可以在 `/usr/local/nagios/etc/objects/` 目录里找到。如果愿意，则可以对里面的对象进行修改以适应你的要求。但是，如果不熟悉如何配置 Nagios，则建议不要这么做。开始时，可以只是按照下面的指令来快速完成监控交换机与路由器的操作。

前提条件

首次监控一台网络交换机时需要对 Nagios 做点额外的工作，记住，仅仅是监控第一台交换机时需要做这些工作。

编辑 Nagios 的主配置文件：

```
vi /usr/local/nagios/etc/nagios.cfg
```

把下面这行最前面的 # 号去掉：



```
#cfg_file=/usr/local/nagios/etc/objects/switch.cfg
```

保存配置文件并退出。

上面的工作是让 Nagios 启用 `/usr/local/nagios/etc/objects/switch.cfg` 这个配置文件里的对象定义。在这个配置文件里可以加一些路由器和交换机的主机与服务对象定义。该配置文件里已经包含几个样例主机、主机组及服务对象定义。对于第一台路由器或交换机，可以只是简单地修改里面已有的主机与服务对象定义，而不用新创建一个。

配置 Nagios

为了监控路由器和交换机，下面要在 Nagios 配置文件里加几个对象定义。

以编辑方式打开 `switch.cfg` 文件：

```
vi /usr/local/nagios/etc/objects/switch.cfg
```

给交换机加一个新的主机对象定义以便监控。如果被监控的是第一台交换机，则可以只修改 `switch.cfg` 文件里的对象定义。修改 `host_name`、`alias` 和 `address` 字段以符合那台交换机的信息：

```
define host{
    use         generic-switch; Inherit default values from a template
    host_name    linksys-srw224p; The name we're giving to this switch
    alias        Linksys SRW224P Switch; A longer name associated with the switch
    address      192.168.1.253; IP address of the switch
    hostgroups   allhosts,switches; Host groups this switch is associated with
}
```

现在，可以在该配置文件里继续添加服务定义，以使 Nagios 监控交换机上的不同方面的内容。如果是第一台交换机，则可以只修改 `switch.cfg` 里的服务对象定义。

增加如下的服务定义以监控 Nagios 主机到交换机的丢包率和 RTA，在一般情况下每 5 分钟检测一次：

```
define service{
    use         generic-service ; Inherit values from a template
    host_name    linksys-srw224p ; The name of the host the service is
associated with
    service_description PING; The service description
    check_command    check_ping!200.0,20%!600.0,60%; The command used to
monitor the service
    normal_check_interval    5; Check the service every 5 minutes under normal
conditions
    retry_check_interval    1; Re-check the service every minute until its
```

```
final/hard state is determined
}
```

这个服务的状态将会处于：

- 紧急（CRITICAL）——条件是 RTA 大于 600ms 或丢包率大于等于 60%；
- 告警（WARNING）——条件是 RTA 大于 200ms 或丢包率大于等于 20%；
- 正常（OK）——条件是 RTA 小于 200ms 或丢包率小于 20%。

如果交换机与路由器支持 SNMP 接口，则可以用 `check_snmp` 插件来监控更丰富的信息。如果不支持 SNMP 接口，则跳过此节内容。在刚才修改的交换机对象定义中加入如下服务定义：

```
define service{
    use          generic-service; Inherit values from a template
    host_name      linksys-srw224p
    service_description Uptime
    check_command   check_snmp!-C public -o sysUpTime.0
}
```

在上述服务定义中的 `check_command` 域里，用 “-C public” 来指定 SNMP 共同体名称为 “public”，用 “-o sysUpTime.0” 指定要检测的 OID。

如果要确保交换机上某个指定端口或接口的状态处于运行状态，则可以在对象定义里加入一段定义：

```
define service{
    use          generic-service; Inherit values from a template
    host_name      linksys-srw224p
    service_description Port 1 Link Status
    check_command   check_snmp!-C public -o ifOperStatus.1 -r 1 -m
RFC1213-MIB
}
```

在上例中，“-o ifOperStatus.1” 指取出交换机的端口编号为 1 的 OID 状态。“-r 1” 是让 `check_snmp` 插件检查返回一个正常（OK）状态，如果在 SNMP 查询结果中存在 “1”，则说明交换机端口处于运行状态，如果没找到 1 就是紧急（CRITICAL）状态。“-m RFC1213-MIB” 是可选的，它告诉 `check_snmp` 插件只加载 “RFC1213-MIB” 库而不是加载每个在系统里的 MIB 库，这样可以加快插件的运行速度。

这就是 SNMP 库的例子。有成百上千种信息可以通过 SNMP 来监控，这完全取决于你需要做什么和如何来做监控。

可以监控交换机或路由器的带宽利用率，用 MRTG 绘图并让 Nagios 在流量超出指定阈值时报警。check_mrtgtraf 插件可以实现该功能，它已经包含在 Nagios 插件软件发行包中。

需要让 check_mrtgtraf 插件知道将 MRTG 数据存入文件的方式，以及相关的阈值信息等。在本例子中，监控了一个 Linksys 交换机。MRTG 日志保存于/var/lib/mrtg/192.168.1.253_1.log 文件中。下面就是用于监控的服务定义，它可以用于监控保存到日志文件中的带宽数据：

```
define service{
    use          generic-service; Inherit values from a template
    host_name    linksys-srw224p
    service_description Port 1 Bandwidth Usage
    check_command
check_local_mrtgtraf!/var/lib/mrtg/192.168.1.253_1.log!AVG!1000000,2000000!5
000000,5000000!10
}
```

在上例中，“/var/lib/mrtg/192.168.1.253_1.log”参数传给 check_local_mrtgtraf 命令的意思是插件的 MRTG 日志文件在这个文件里读写，“AVG”参数的意思是取带宽的统计平均值，“1000000,2000000”参数指流入的告警阈值（以字节为单位），“5000000,5000000”是输出流量紧急状态阈值（以字节为单位），“10”指如果 MRTG 日志超过 10 分钟没有数据，则返回一个紧急状态（正常情况下应该每 5 分钟更新一次）。

保存一下配置文件。

重启 Nagios

如果已经修改好 Nagios 配置文件，则需要验证你的配置文件并重启 Nagios。

如果验证配置文件过程中有什么错误信息，则在做下一步前一定要修正好配置文件。一定要保证验证过程中不再有出错信息再启动或重启 Nagios。

4. 监控网络打印机

Nagios 可以监控网络打印机。特别是有内置或外置 JetDirect 卡/设备的 HP（惠普）打印机，或其他（像 Troy PocketPro 100S 或 Netgear PS101）支持 JetDirect 协议的打印机。

check_hpjd 插件（该插件是 Nagios 插件软件发行包的标准组成部分）可以用 SNMP 方式来监控 JetDirect 兼容型打印机。该插件可以检查打印机的如下状态：

- 卡纸；
- 无纸；
- 打印机离线；
- 需要人工干预；

- 墨盒墨粉低；
- 内存不足；
- 开外壳；
- 输出托盘已满；
- 其他。

概览

监控网络打印机的状态很简单。有 JetDirect 功能的打印机一般提供 SNMP 功能，可以用 `check_hpjd` 插件来检测状态。

`check_hpjd` 插件只有当系统中安装 `net-snmp` 和 `net-snmp-utils` 软件包时才会被编译和安装。要保证在 `/usr/local/nagios/libexec` 目录下有 `check_hpjd` 文件，否则，要安装好 `net-snmp` 和 `net-snmp-utils` 软件包再重新编译安装 Nagios 插件包。监控网络打印机的示意图如图 6-5 所示。

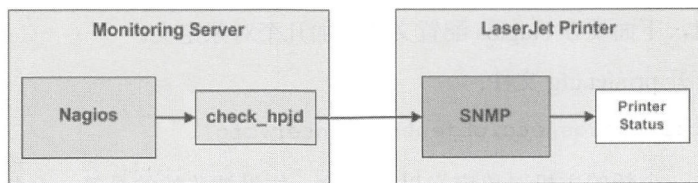


图 6-5 监控网络打印机的示意图

步骤

为完成打印机的监控，有几个步骤要做：

- 确认一下前提条件；
- 给打印机创建要监控的主机与服务对象定义；
- 重启动 Nagios 守护进程。

为使整个过程简单化，先完成少量配置文件的工作：

- 把 `check_hpjd` 命令加入 `commands.cfg` 文件，可以用 `check_hpjd` 插件来监控打印机；
- 一个打印机的主机对象模板（命名为 `generic-printer`）已经在 `templates.cfg` 文件里创建好了，这样就可以更容易地加入一个新的打印机的主机对象定义。

常用的配置文件可以在 `/usr/local/nagios/etc/objects/` 目录里找到。可以对里面的对象进行修改以适应你的要求。但是，如果不熟悉如何配置 Nagios，则建议不要这么做。对于新手来说，只要按照下面的指令操作来快速完成监控打印机的操作即可。

前提条件

首次监控一台打印机时需要对 Nagios 做点额外的工作，记住，仅仅是监控第一台打印机时

需要做这些工作。

编辑 Nagios 的主配置文件：

```
vi /usr/local/nagios/etc/nagios.cfg
```

把下面这行最前面的“#”号去掉：

```
#cfg_file=/usr/local/nagios/etc/objects/printer.cfg
```

保存配置文件并退出。

上面的工作是让 Nagios 启用/usr/local/nagios/etc/objects/printer.cfg 这个配置文件里的对象定义。在这个配置文件里可以加一些打印机的主机与服务对象定义。该配置文件里已经包含几个主机、主机组及服务对象定义的样例。对于第一台打印机，可以只是简单地修改里面已有的主机与服务对象定义而不用新创建一个。

配置 Nagios

为监控打印机，下面要在 Nagios 配置文件里加几个对象定义。

以编辑方式打开 printer.cfg 文件：

```
vi /usr/local/nagios/etc/objects/printer.cfg
```

给打印机添加一个新的主机对象定义以便监控。如果被监控的是第一台打印机，则可以只修改 printer.cfg 文件里的对象定义。修改 host_name、alias 和 address 字段以符合那台打印机：

```
define host{
    use         generic-printer; Inherit default values from a template
    host_name    hplj2605dn; The name we're giving to this printer
    alias        HP LaserJet 2605dn; A longer name associated with the printer
    address      192.168.1.30; IP address of the printer
    hostgroups   allhosts; Host groups this printer is associated with
}
```

现在可以在该配置文件里继续添加服务定义，以使 Nagios 监控打印机上的不同方面的内容。如果是第一台打印机，则可以只修改 printer.cfg 里的服务对象定义。

增加如下的服务定义以监控打印机的状态。服务用 check_hpjd 插件来检测打印机状态，默认情况下每 10 分钟检测一次：

```
define service{
    use         generic-service; Inherit values from a template
    host_name    hplj2605dn; The name of the host the service is associated with
    service_description Printer Status; The service description
    check_command check_hpjd!-C public; The command used to monitor the
```

```
service
    normal_check_interval 10; Check the service every 10 minutes under
normal conditions
    retry_check_interval 1; Re-check the service every minute until its
final/hard state is determined
}
```

加入一个默认每 10 分钟进行一次的 PING 检测服务，用于检测 RTA、丢包率和网络连接状态：

```
define service{
    use                generic-service
    host_name          hplj2605dn
    service_description PING
    check_command       check_ping!3000.0,80%!5000.0,100%
    normal_check_interval 10
    retry_check_interval 1
}
```

保存一下配置文件。

重启 Nagios

步骤同之前的一样，修改好 Nagios 配置文件后，先验证你的配置文件，而后再重启 Nagios。

5. 监控公众化服务

所谓“公众化”服务是指在网络里常见的服务——不管本地网络还是因特网。公众服务包括 HTTP、POP3、IMAP、FTP 和 SSH。其实在日常使用中还有更多的基础服务。这些服务与应用，包括所依托的协议，可以被 Nagios 直接监控而不需要额外的支持。

与之相对的是私有服务，如果没有某些中间件来做代理，则 Nagios 是无法监控的。主机上的私有服务包括 CPU 负荷、内存占用率、磁盘利用率、当前登录用户、进程信息等。这些私有服务或属性不能暴露给外部客户。这样就要在这些被监控的主机上安装一些中间件做代理以取得此类信息。

监控 HTTP

check_http 插件可以监控 HTTP。该插件可以透过 HTTP 监控响应时间、错误代码、返回页面上的字符串、服务器证书和其他更多的东西。

在 commands.cfg 文件里包含一个使用 check_http 插件的命令定义，如下：

```
define command{
```

```

name          check_http;
command_name   check_http;
command_line   $USER1$/check_http -I $HOSTADDRESS$ $ARG1$;
}

```

监控在远程机器上的 HTTP 服务的简单的服务对象定义示例如下：

```

define service{
    use      generic-service;
    host_name    remotehost;
    service_description HTTP;
    check_command  check_http;
}

```

上面定义的含义是，如果 Web 服务在 10 秒内没有响应将产生警报或返回一个 HTTP 的错误码（403、404 等），同时会产生警报。

下面看看对 HTTP 服务更高级的监控。这个服务定义将会检测如下内容：在/download/index.php 页面里是否包含“latest-version.tar.gz”字符串，如果没有指定字符串，或 URI 非法，又或在 5 秒内没有响应，则它将产生一个错误。

```

define service{
    use      generic-service;
    host_name    remotehost;
    service_description Product Download Link;
    check_command  check_http!-u /download/index.php -t 5 -s "latest-
version.tar.gz";
}

```

监控 FTP 服务器

check_ftp 插件可以监控 FTP 服务器。在 commands.cfg 文件里包含一个使用 check_ftp 插件的命令定义：

```

define command{
    command_name   check_ftp;
    command_line   $USER1$/check_ftp -H $HOSTADDRESS$ $ARG1$;
}

```

监控在远程机器上的 FTP 服务器的简单的服务对象定义示例如下：

```

define service{
    use      generic-service;

```



```
host_name      remotehost;
service_description FTP;
check_command  check_ftp;
}
```

这个服务对象定义会在 FTP 服务器 10 秒内不响应时给出一个警报。

下面对 FTP 服务器做一些高级监控。下面这个服务将检测 FTP 服务器绑定在远程主机上的 1023 端口的 FTP 服务。如果在 5 秒内没有响应或服务里没有回应字符串 “Pure-FTPd [TLS]”，将产生警报。

```
define service{
    use      generic-service;
    host_name      remotehost;
    service_description Special FTP;
    check_command  check_ftp!-p 1023 -t 5 -e "Pure-FTPd [TLS]";
}
```

监控 SSH 服务器

check_ssh 插件可以监控 SSH 服务器。在 commands.cfg 文件里包含一个使用 check_ssh 插件的命令定义，如下：

```
define command{
    command_name      check_ssh;
    command_line      $USER1$/check_ssh $ARG1$ $HOSTADDRESS$;
}
```

监控在远程机器上的 SSH 服务器的简单的服务对象定义示例如下：

```
define service{
    use      generic-service;
    host_name      remotehost;
    service_description SSH;
    check_command  check_ssh;
}
```

这个服务对象定义将会在 SSH 服务 10 秒内不响应时给出一个警报。

下面对 SSH 服务器做一些高级监控。下面这个服务将检测 SSH 服务，如果 5 秒内不响应或服务器版本串里没有能匹配的字符串 “OpenSSH_4.2”，将产生一个警报。

```
define service{
    use      generic-service;
```

```

host_name      remotehost;
service_description SSH Version Check;
check_command  check_ssh!-t 5 -r "OpenSSH_4.2";
}

```

监控 SMTP 服务器

check_smtp 插件可以监控 SMTP 服务器。在 commands.cfg 文件里包含一个使用 check_smtp 插件的命令定义，如下：

```

define command{
    command_name    check_smtp;
    command_line    $USER1$/check_smtp -H $HOSTADDRESS$ $ARG1$;
}

```

监控在远程机器上的 SMTP 服务器的简单的服务对象定义示例如下：

```

define service{
    use      generic-service;
    host_name      remotehost;
    service_description SMTP;
    check_command  check_smtp;
}

```

这个服务对象定义将在 SMTP 服务器 10 秒内不响应时给出一个警报。

下面是更高级的监控。下面这个服务将检测 SMTP 服务，如果服务在 5 秒内不响应或返回串里没有“mygreatmailserver.com”，将产生一个警报。

```

define service{
    use      generic-service;
    host_name      remotehost;
    service_description SMTP Response Check;
    check_command  check_smtp!-t 5 -e "mygreatmailserver.com";
}

```

监控 POP3 服务器

check_pop 插件可以监控 POP3 服务器。在 commands.cfg 文件里包含一个使用 check_pop 插件的命令定义，如下：

```

define command{
    command_name    check_pop;
}

```

```
command_line    $USER1$/check_pop -H $HOSTADDRESS$ $ARG1$;
}
```

监控在远程机器上的 POP3 服务器的简单的服务对象定义示例如下：

```
define service{
    use        generic-service;
    host_name    remotehost;
    service_description POP3;
    check_command    check_pop;
}
```

这个服务对象定义将在 POP3 服务 10 秒内不响应时给出一个警报。

下面是更高级的监控。下面这个服务将检测 POP3 服务，当服务在 5 秒内不响应或返回串里没有“mygreatmailserver.com”时，将产生一个警报。

```
define service{
    use        generic-service;
    host_name    remotehost;
    service_description POP3 Response Check;
    check_command    check_pop!-t 5 -e "mygreatmailserver.com";
}
```

监控 IMAP 服务器

check_imap 插件可以监控 IMAP 服务器。在 commands.cfg 文件里包含一个使用 check_imap 插件的命令定义，如下：

```
define command{
    command_name    check_imap;
    command_line    $USER1$/check_imap -H $HOSTADDRESS$ $ARG1$;
}
```

监控在远程机器上的 IMAP 服务器的简单的服务对象定义示例如下：

```
define service{
    use        generic-service;
    host_name    remotehost;
    service_description IMAP;
    check_command    check_imap;
}
```

这个服务对象定义将在 IMAP 服务 10 秒内不响应时给出一个警报。

下面是更高级的监控。下面这个服务将检测 IMAP 服务，当服务在 5 秒内不响应或返回串里没有“mygreatmailserver.com”时将产生一个警报。

```
define service{
    use         generic-service;
    host_name    remotehost;
    service_description IMAP4 Response Check;
    check_command check_imap!-t 5 -e "mygreatmailserver.com";
}
```

重启 Nagios

步骤同之前的一样，修改好 Nagios 配置文件后，需要先验证你的配置文件，再重启 Nagios。

6.2.2 Nagios 插件

正如上一节“Nagios 监控”所述，Nagios 通过各种外部构件（插件）来实现对各种设备、状态进行监控的目的。插件作为被监控对象与 Nagios 之间的代理，承担着各种监控“脏活（dirty work）”。

1. 什么是插件

插件是编译的执行文件或脚本（Perl 脚本、Shell 脚本等），可以在命令行下执行对主机或服务状态检查。Nagios 运行这些插件的检测结果来决定网络中的主机和服务的当前状态。

当需要检测主机或服务状态时，Nagios 总是执行一个插件程序，插件来完成检查并输出简洁的结果给 Nagios。Nagios 将处理这些来自插件的结果并针对这些结果做出响应动作，比如运行事件处理句柄、发出告警等。

2. 插件是一个抽象层

插件是一个位于 Nagios 守护程序里的监控逻辑和实际被监控的主机与服务之间的抽象层。

Nagios 插件架构如图 6-6 所示。

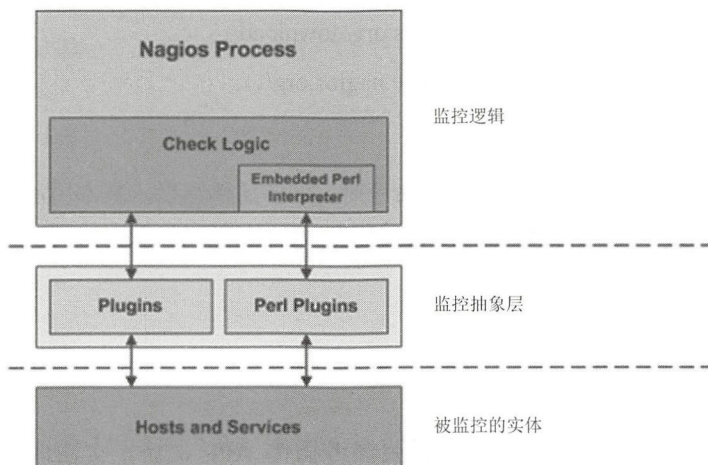


图 6-6 Nagios 插件架构

在插件构架之上可以监控所有想要监控的东西。如果能自动地处理检测过程，则可以用 Nagios 来监控它。Nagios 已经有很多插件用于监控基础性资源，比如 CPU 负荷、磁盘利用率、ping 包率等，如果想监控别的，则需要查阅插件 API 文档来进行插件的开发。

在插件构架之下，事实上 Nagios 并不需要了解所要监控的对象。可以监控网络流量状态、数据错包率、房间温度、CPU 电压值、风扇转速、处理器负载、磁盘空间，或早上起来你的面包机烤出的面包的色泽，Nagios 不会去理解这些被监控的对象，它只是忠实地记录这些被管理资源的状态变化轨迹。只有插件自己知道监控了什么东西，并如何完成检测。

3. 可用插件

有许多插件可用于监控不同的设备和服务：

- HTTP、POP3、IMAP、FTP、SSH、DHCP；
- CPU 负荷、磁盘利用率、内存占用率、当前用户；
- UNIX/Linux、Windows 和 Netware 服务器；
- 路由器和交换机；
- 等等。

4. 获得插件

插件不与 Nagios 包一起发布，但可以下载 Nagios 官方插件和由 Nagios 用户编写并维护的额外插件。

- Nagios Plugins 工程：<http://nagiosplug.sourceforge.net/>；


```
-e MYSQL_PASSWORD="zabbix_pwd" \
-e MYSQL_ROOT_PASSWORD="root_pwd" \
-d mysql:5.7
```

(2) 开启一个 Zabbix Java gateway 实例:

```
# docker run --name zabbix-java-gateway -t \
  -d zabbix/zabbix-java-gateway:latest
```

(3) 开启 Zabbix server 实例, 并链接到创建的 MySQL server 实例:

```
# docker run --name zabbix-server-mysql -t \
  -e DB_SERVER_HOST="mysql-server" \
  -e MYSQL_DATABASE="zabbix" \
  -e MYSQL_USER="zabbix" \
  -e MYSQL_PASSWORD="zabbix_pwd" \
  -e MYSQL_ROOT_PASSWORD="root_pwd" \
  -e ZBX_JAVAGATEWAY="zabbix-java-gateway" \
  --link mysql-server:mysql \
  --link zabbix-java-gateway:zabbix-java-gateway \
  -p 10051:10051 \
  -d zabbix/zabbix-server-mysql:latest
```

(4) 开启 Zabbix Web 实例, 并链接到创建的 MySQL server 和 Zabbix server 实例:

```
# docker run --name zabbix-web-nginx-mysql -t \
  -e DB_SERVER_HOST="mysql-server" \
  -e MYSQL_DATABASE="zabbix" \
  -e MYSQL_USER="zabbix" \
  -e MYSQL_PASSWORD="zabbix_pwd" \
  -e MYSQL_ROOT_PASSWORD="root_pwd" \
  --link mysql-server:mysql \
  --link zabbix-server-mysql:zabbix-server \
  -p 80:80 \
  -d zabbix/zabbix-web-nginx-mysql:latest
```

示例 2

本示例演示如何运行 PostgreSQL 版本的 Zabbix server, 其 Zabbix Web 界面基于 Nginx Web 服务器和 snmptraps。

(1) 开启一个空的 PostgreSQL server 实例:

```
# docker run --name postgres-server -t \
```

```
-e POSTGRES_USER="zabbix" \  
-e POSTGRES_PASSWORD="zabbix" \  
-e POSTGRES_DB="zabbix_pwd" \  
-d postgres:latest
```

（2）开启一个 Zabbix snmptraps 实例：

```
# docker run --name zabbix-snmptraps -t \  
-v ./zbx_instance/snmptraps:/var/lib/zabbix/snmptraps:rw \  
-v /var/lib/zabbix/mibs:/usr/share/snmp/mibs:ro \  
-p 162:162/udp \  
-d zabbix/zabbix-snmptraps:latest
```

（3）开启 Zabbix server 实例，并链接到创建的 PostgreSQL server 实例：

```
# docker run --name zabbix-server-mysql -t \  
-e DB_SERVER_HOST="postgres-server" \  
-e POSTGRES_USER="zabbix" \  
-e POSTGRES_PASSWORD="zabbix" \  
-e POSTGRES_DB="zabbix_pwd" \  
-e ZBX_ENABLE_SNMP_TRAPS="true" \  
--link postgres-server:postgres \  
-p 10051:10051 \  
--volumes-from zabbix-snmptraps \  
-d zabbix/zabbix-server-mysql:latest
```

（4）开启 Zabbix Web 实例，并链接到创建的 PostgreSQL server 和 Zabbix server 实例：

```
# docker run --name zabbix-web-nginx-pgsql -t \  
-e DB_SERVER_HOST="postgres-server" \  
-e POSTGRES_USER="zabbix" \  
-e POSTGRES_PASSWORD="zabbix" \  
-e POSTGRES_DB="zabbix_pwd" \  
--link postgres-server:postgres \  
--link zabbix-server-mysql:zabbix-server \  
-p 443:443 \  
-v /etc/ssl/nginx:/etc/ssl/nginx:ro \  
-d zabbix/zabbix-web-nginx-pgsql:latest
```


6.3.2 Zabbix 的基本概念

下面介绍 Zabbix 的一些基本术语和核心概念。

1. 基本术语

- host（主机）——想监控的网络设备（需要知道 IP/DNS）。
- host group（主机组）——逻辑的主机组，包含 host 和 template。在同一个 host group 里面的 host 和 template 不能互相链接。host group 通常用于给不同的用户组创建访问权限。
- item（监控项）——从主机中收集到的数据特定部分。
- trigger（触发器）——一个逻辑表达式，用来表达从监控项获取的数据达到了预设的问题阈值。当收到的监控值达到了预设的阈值，则触发器状态由“OK”变更为“Problem”；当收到的监控值低于阈值时，则状态保持/变更为“OK”。
- event（事件）——一个事件发生，例如 trigger 状态变更或一个 discovery/agent 自动注册等。
- problem——trigger 处于“Problem”状态。
- action（动作）——当一个 event 发生时预设的处理过程。一个 action 包括操作（如发送告警）和条件（当操作被执行）。
- escalation——一个 action 中执行操作的自定义场景；发送 notification/执行 remote command 的序列。
- media（媒介）——发送通知的渠道。
- notification（通知）——通过 media 渠道发送事件的消息。
- remote command（远程命令）——当监控主机达到某些条件后预设的自动执行的命令。
- template（模板）——一组包含 item、trigger、绘图、面板（screen）、application、低级别自动发现规则、Web scenario 等且能被其他主机应用的实体。模板能够提升主机部署监控任务的速度，同时也非常容易对监控任务做批量更新。模板可以被直接链接到独立主机。
- application（应用）——item 的逻辑分组。
- Web scenario（Web 方案）——对一个 Web 站点的可用性进行检查的一个或多个 HTTP 请求。
- frontend（前端）——Zabbix 提供的 Web 界面。
- Zabbix API——Zabbix API 允许通过 JSON RPC 协议创建、更新、获得 Zabbix 对象（如

host、item、绘图等），以及完成自定义任务。

- Zabbix server——Zabbix 软件中心进程，用于连通 Zabbix proxy 及 agent 完成监控、计算 trigger、发送 notification，以及承担数据的集中存储的功能。
- Zabbix agent——部署在监控主机上的进程，用于监控本地资源和应用。
- Zabbix proxy——替代 Zabbix server 完成数据收集的进程，通常用于降低中心 Zabbix server 的负载。

2. server

Zabbix server 是 Zabbix 软件的中心进程，其执行 polling 和 trapping 来采集数据，评估是否触发触发器，发送通知给用户。这是 Zabbix agent 和 proxy 报告系统可用性和完整性数据的中心组件。server 也可以通过简单服务检查来完成远程网络服务检测，比如 Web 服务器、邮件服务器。

server 是所有配置、统计和操作数据的中心存储仓库，也在所有的监控系统中扮演故障发生时通知管理员的角色。

基础 Zabbix server 依据功能不同划分为三个部分，分别为 Zabbix server、web frontend 及数据库。

由于 Zabbix 所有的配置信息都保存在数据库中，server 和 web frontend 可以直接进行操作。比如，通过 web frontend（或者 API）创建一个新的 item 时，它将创建的数据插入数据库。大约一分钟后，Zabbix server 会查询监控项数据表，并将查询的 item 列表保存在自己的缓存中。这也是为什么通过 Zabbix frontend 进行的变更将在两分钟左右生效的原因。

server 进程

Zabbix server 以守护进程方式运行，server 可以通过以下命令启动：

```
shell> cd sbin
shell> ./zabbix_server
```

也可以在启动 Zabbix server 时使用下面的命令行参数：

-c --config <file>	配置文件的绝对路径（默认是 /etc/zabbix/zabbix_server.conf）
-R --runtime-control <option>	运行管理命令
-h --help	显示本帮助
-V --version	显示版本号

注：OpenBSD 和 NetBSD 平台不支持 runtime control（运行时控制）。

下面是使用参数的例子：

```
shell> zabbix_server -c /usr/local/etc/zabbix_server.conf
shell> zabbix_server --help
shell> zabbix_server -V
```

下面是使用 runtime control 来重载 server 配置缓存的例子：

```
shell> zabbix_server -c /usr/local/etc/zabbix_server.conf -R config_cache_reload
```

下面是使用 runtime control 来触发 housekeeper 执行的例子：

```
shell> zabbix_server -c /usr/local/etc/zabbix_server.conf -R housekeeper_execute
```

下面是使用 runtime control 来修改日志级别的例子：

Increase log level of all processes:

```
shell> zabbix_server -c /usr/local/etc/zabbix_server.conf -R log_level_increase
```

Increase log level of second poller process:

```
shell> zabbix_server -c /usr/local/etc/zabbix_server.conf -R log_level_increase=poller, 2
```

Increase log level of process with PID 1234:

```
shell> zabbix_server -c /usr/local/etc/zabbix_server.conf -R log_level_increase=1234
```

Decrease log level of all http poller processes:

```
shell> zabbix_server -c /usr/local/etc/zabbix_server.conf -R log_level_decrease="http poller"
```

进程账号

Zabbix 设计运行在非 root 账号下。它可以运行在任何非 root 账号下。因此 Zabbix server 可以运行在非 root 账号上而无须担心有问题。

如果想让 server 运行在 root 账号下，必须在系统中调整已经默认写死的“zabbix”用户，并相应地修改“AllowRoot”参数。

如果 Zabbix server 和 agent 运行在同一台机器上，建议分别运行在不同的用户下，因为一旦运行在同一个用户下，agent 就可以访问 server 的配置文件，并且能够轻松取得 Zabbix Admin 级别用户的信息，例如数据库密码。

启动脚本

该类脚本用于在系统启动/关闭时自动启动/关闭 Zabbix 进程，位于 misc/init.d 目录下。

支持的平台

由于安全需求和服务器操作的关键任务特性，UNIX 是唯一的操作系统，能够持续提供必要的性能、容错能力和应变能力。Zabbix 可以在市面上最新的 UNIX 操作系统上运行。

Zabbix server 在以下平台上进行了测试：Linux、Solaris、AIX、HP-UX、Mac OS X、FreeBSD、OpenBSD、NetBSD、SCO Open Server、Tru64/OSF1。

Zabbix 可以在其他类 UNIX 操作系统下正常工作。

3. agent

Zabbix agent 部署在被监控机器上，用来监控本地资源和应用（如硬盘、内存、处理器统计等）。

Zabbix agent 聚合本地的运行信息并将数据发送给 Zabbix server 以进一步处理。一旦出现异常（比如硬盘空间满或服务宕掉），Zabbix server 将发送告警给管理员，报告本次异常。

Zabbix agent 利用本地系统调用完成统计信息收集，因此它非常高效。

被动（passive）和主动（active）检查

Zabbix agent 提供被动和主动两种检查方式。

在被动检查方式中，agent 应答数据请求，Zabbix server 或 proxy 询问 agent 数据，如 CPU 负荷情况，然后 Zabbix agent 返回结果给 server。

主动检查处理过程将相对复杂。agent 必须首先进行一次请求 Zabbix server 索取 item 列表，然后发送对应的值给 server。

无论被动还是主动检查，都需要在监控项类型中选择“Zabbix agent”或“Zabbix agent (active)”。

支持的平台

Zabbix agent 支持的平台：Linux、IBM AIX、FreeBSD、NetBSD、OpenBSD、HP-UX、Mac OS X、Solaris（9、10、11）、Windows（支持所有 Windows 2000 以后的桌面和服务器版本）。

运行 agent 进程

Zabbix agent 运行在被监控 host 上，可以通过守护进程的方式运行。

通过以下命令在 UNIX 上运行 Zabbix agent：

```
shell> cd sbin
shell> ./zabbix_agentd
```

运行 Zabbix agent 时可以指定如下命令行参数：

```
-c --config <config-file>      指定配置文件，UNIX 默认的为 /usr/local/etc/
```


zabbix_agentd.conf, Windows 默认的为 c:\zabbix_agentd.conf

-h --help	显示本帮助
-V --version	显示 agent 版本号
-p --print	显示已知的监控项并退出
-t --test <item key>	测试指定的监控项并退出

例如, 想查询帮助信息, 则可以执行:

```
shell> zabbix_agentd -h
```

使用命令行参数的例子:

```
shell> zabbix_agentd --print
shell> zabbix_agentd -t "mysql.ping" -c /etc/zabbix/zabbix_agentd.conf
shell> zabbix_agentd.exe -i
shell> zabbix_agentd.exe -i -m -c zabbix_agentd.conf
```

下面是使用 runtime control 修改日志级别的例子:

```
shell> zabbix_agentd -R log_level_increase
shell> zabbix_agentd -R log_level_increase=listener,2
shell> zabbix_agentd -R log_level_increase=1234
shell> zabbix_agentd -R log_level_decrease="active checks"
```

进程账号

同运行 server 类似, 可以在非 root 账号下运行 agent。

如果想让 agent 运行在 root 账号下, 则必须在系统中调整已经默认写死的“zabbix”用户, 并相应地修改 agent 配置的“AllowRoot”参数。

退出代码

2.2 版本之前 Zabbix agent 在成功退出的情况下返回 0, 在故障的情况下返回 255。2.2 及更高版本的 Zabbix agent 在成功退出的情况下返回 0, 在故障的情况下返回 1。

4. proxy

Zabbix proxy 通常用于替代 server 完成对多个监控设备的监控信息收集工作并将数据发送给 Zabbix server。收集的所有数据会先存储在 proxy 本地缓存中, 然后传送给 Zabbix server。

proxy 是可选的, 使用它可以有效地降低在分布式环境中单一的 Zabbix server 负载。通过 proxy 收集数据, server 可以有效降低 CPU 和磁盘 I/O 的消耗。

Zabbix proxy 可以出色地完成远程区域、分支机构、没有本地管理员的网络的集中监控。

Zabbix proxy 使用独立的数据库。

注：Zabbix proxy 数据库可以使用 SQLite、MySQL、PostgreSQL。使用 Oracle 或 IBM DB2 在低等级自动发现规则时存在限制和风险，详情见 https://www.zabbix.com/documentation/3.2/manual/distributed_monitoring/proxies。

proxy 进程

Zabbix proxy 以守护进程方式运行，proxy 可以通过以下命令启动：

```
shell> cd sbin
shell> ./zabbix_proxy
```

也可以在启动 Zabbix proxy 时使用下面的命令行参数：

-c --config <file>	配置文件的绝对路径（默认是 /etc/zabbix/zabbix_server.conf）
-R --runtime-control <option>	运行管理命令
-h --help	显示本帮助
-V --version	显示版本号

注：在 OpenBSD 和 NetBSD 平台，不支持 runtime control。

下面是使用参数的例子：

```
shell> zabbix_proxy -c /usr/local/etc/zabbix_proxy.conf
shell> zabbix_proxy --help
shell> zabbix_proxy -V
```

下面是使用 runtime control 来重载 proxy 配置缓存的例子：

```
shell> zabbix_proxy -c /usr/local/etc/zabbix_proxy.conf -R config_cache_reload
```

下面是使用 runtime control 来触发 housekeeper 执行的例子：

```
shell> zabbix_proxy -c /usr/local/etc/zabbix_proxy.conf -R housekeeper_execute
```

下面是使用 runtime control 来修改日志级别的例子：

Increase log level of all processes:

```
shell> zabbix_proxy -c /usr/local/etc/zabbix_proxy.conf -R log_level_increase
```

Increase log level of second poller process:

```
shell> zabbix_proxy -c /usr/local/etc/zabbix_proxy.conf -R log_level_increase=poller,2
```

```
Increase log level of process with PID 1234:
```

```
shell> zabbix_proxy -c /usr/local/etc/zabbix_proxy.conf -R log_level_
increase=1234
```

```
Decrease log level of all http poller processes:
```

```
shell> zabbix_proxy -c /usr/local/etc/zabbix_proxy.conf -R log_level_
decrease="http poller"
```

进程账号

同运行 server 类似，可以在非 root 账号下运行 proxy。

如果想让 proxy 运行在 root 账号下，则必须在系统中调整已经默认写死的“zabbix”用户，并相应地修改 proxy 配置的“AllowRoot”参数。

支持的平台

Zabbix proxy 所支持的平台与 Zabbix server 一致。

5. Java gateway

自 Zabbix 2.0 开始，被称为“Zabbix Java gateway”的新增 Zabbix 守护进程提供了原生对 JMX 应用的监控。Zabbix Java gateway 是采用 Java 编写的一个守护进程。当 Zabbix server 想知道主机 JMX 计数器的值时，它请求 Zabbix Java gateway，Zabbix Java gateway 利用 JMX management API 请求远程的有关应用。应用不需要额外安装软件，只需要在启动时在命令行指定-Dcom.sun.management.jmxremote 选项即可。

Java gateway 接受来自 Zabbix server 或 proxy 的连接，只能使用“passive proxy”的方式。在 Zabbix server 或 proxy 的配置文件中要指定需要访问的 Java gateway，因此在每一个 Zabbix server 或 proxy 中只能配置一个 Java gateway。如果一个主机有 JMX agent 及其他类型的 item，则只有 JMX agent 类型的 item 可以通过 Java gateway 进行监控。

当在 Java gateway 上的一个 item 值更新了，Zabbix server 或 proxy 将连接 Java gateway 来请求该值，Java gateway 将获取的值传回 server 或 proxy。同样，Java gateway 不会缓存任何值。

Zabbix server 或 proxy 可以通过 StartJavaPollers 来控制连接 Java gateway 的进程。在内部，Java gateway 通过 START_POLLERS 控制选项来启动多线程。在服务端，如果一个连接请求超过了 Timeout 设定的秒数，则连接会终止，但 Java gateway 也许此时依然会从 JMX 计数器中检索该值。因此，为了解决这个问题，Java gateway 会有一个 TIMEOUT 选项用于设置 JMX 网络操作超时。

为了更好地提升单个连接的性能，Zabbix server 或 proxy will 会尽量将发往同一个 JMX 目标的请求一起发送到 Java gateway（这个跟 item 的时间间隔有关）。

建议 StartJavaPollers 小于或等于 START_POLLERS，否则可能导致当请求访问 Java gateway



时，Java gateway 没有多余的线程进行处理。

以下内容将详细讲解如何安装和运行 Zabbix Java gateway，如何配置 Zabbix server（或 proxy）利用 Zabbix Java gateway 完成 JMX 监控，如何在 Zabbix GUI 中配置 JMX 计数器 item。

获取 Java gateway

有两种方式可以得到 Java gateway：一种是通过 Zabbix 官网下载 Java gateway 包，地址为 <http://www.zabbix.com/download.php>；另一种是通过源码编译 Java gateway。

为了编译 Java gateway，需要在运行 `./configure` 时指定 `--enable-java` 选项。建议在安装时指定 `--prefix` 选项而不使用默认的 `/usr/local`，因为在安装 Java gateway 时将创建整个目录树，而非单一的可执行文件：

```
$ ./configure --enable-java --prefix=$PREFIX
```

使用 `make` 完成 Java gateway 编译并打包成一个 JAR 文件。注意本步骤将使用 `javac` 和 `jar`，因此需要保证它们在 `path` 中：

```
$ make
```

现在将在 `src/zabbix_java/bin` 下得到 `zabbix-java-gateway-$VERSION.jar` 文件。如果想在指定的目录下轻松地使用 Java gateway 以完成配置和运行，请确保有足够的权限运行 `make install`：

```
$ make install
```

最终在 `$PREFIX/sbin/zabbix_java` 下会看到 shell 脚本、JAR 包和配置文件的用途：

```
bin/zabbix-java-gateway-$VERSION.jar
```

Java gateway JAR 文件：

```
lib/logback-core-0.9.27.jar
lib/logback-classic-0.9.27.jar
lib/slf4j-api-1.6.3.jar
lib/android-json-4.3_r3.1.jar
```

Java gateway 依赖 Logback、SLF4J 和 Android JSON 库：

```
lib/logback.xml
lib/logback-console.xml
```

Logback 的配置文件：

```
shutdown.sh
startup.sh
```

可以通过脚本方便地启动/关闭 Java gateway：

```
setting.sh
```




配置和运行 Java gateway

默认情况下, Java gateway 监听 10052 端口。如果计划使用不同的端口来运行 Java gateway, 则需要通过 setting.sh 脚本指定相应的端口。读者可以自行参阅配置说明 https://www.zabbix.com/documentation/3.2/manual/appendix/config/zabbix_java。

一旦完成了配置, 就可以通过 startup 脚本来启动 Java gateway:

```
$ ./startup.sh
```

如果不需要 Java gateway, 则可以运行 shutdown 脚本关闭它:

```
$ ./shutdown.sh
```

不像 server 和 proxy, Java gateway 是轻量级的, 并不需要数据库支持。

配置 server 使用 Java gateway

当前 Java gateway 已经运行, 接下来需要告诉 Zabbix server 如何找到 Zabbix Java gateway。因此需要在 server 配置文件中指定 JavaGateway 及 JavaGatewayPort 参数。如果 JMX 应用采用 Zabbix proxy 进行监控, 则需要在 proxy 配置文件中指定对应的连接参数。

```
JavaGateway=192.168.3.14
JavaGatewayPort=10052
```

默认情况下, server 并不会派生出任何进程去进行 JMX 监控。如果想完成 JMX 监控, 则需要指定预派生出来的 Java pollers 进程数, 也可通过类的方式指定常见的 pollers 和 trappers。

```
StartJavaPollers=5
```

在完成配置后, 千万不要忘记重启 server (或 proxy)。

在 Java gateway 中进行调试

万一 Java gateway 出现了若干问题, 在前段可以看到的监控项报错信息并不充分, 也可以通过查看 Java gateway 日志文件来获得更多信息。

默认情况下, Java gateway 将日志记录到 tmp/zabbix_java.log 文件中, 日志级别为 “info”。有时根据调试的需要, 可以修改级别为 “debug”。可以通过修改 lib/logback.xml 将 <root> 标签更改为 “debug”:

```
<root level="debug">
  <appender-ref ref="FILE" />
</root>
```

需要注意的是, 并不像 Zabbix server 或 proxy 那样, 修改完 logback.xml 并不需要重启 Zabbix Java gateway。修改后的配置将会自动被加载。当完成调试之后, 可以将日志级别替换为 “info”。



如果想将日志记录到其他文件或以其他方式完成存储（如数据库），则按照需求调整 `logback.xml` 文件即可，配置详情可自行参阅 Logback 手册，地址为 <http://logback.qos.ch/manual/>。

有时为了方便调试，不想采用守护进程方式而想采用控制台方式，需要注释掉 `setting.sh` 脚本中的 `PID_FILE` 参数。一旦没有找到 `PID_FILE` 参数，`startup.sh` 脚本将直接以控制台方式运行。Logback 也将使用 `lib/logback-console.xml` 文件，日志不仅记录到控制台，其级别也将变更为“debug”。

最后请注意，由于 Java gateway 使用 SLF4J 作为日志框架，所以可以选择任意日志框架的 JAR 文件在 `lib` 目录中替换 Logback。详情可以参见 SLF4J 手册，地址为 <http://www.slf4j.org/manual.html>。

6. sender

Zabbix sender 命令行工具常用于发送性能数据给 Zabbix server。

该工具常用于在长时间运行的用户的自定义脚本中，以便不断发送可用性及性能数据。

运行 Zabbix sender

在 UNIX 下运行 Zabbix sender 的例子：

```
shell> cd bin
shell> ./zabbix_sender -z zabbix -s "Linux DB3" -k db.connections -o 43
```

参数的对应说明：

- `z`——指定 Zabbix server 的 host（常用 IP 地址）；
- `s`——监控 host 名称（在 Zabbix 前段中填写的主机名）；
- `k`——item 键名；
- `o`——需要发送的值。

Zabbix sender 可以在一个 input 文件中发送多个值，详情可以参阅 https://www.zabbix.com/documentation/3.2/manpages/zabbix_sender。

不管在类 UNIX 系统还是在 Windows 系统中，Zabbix sender 都接收非 BOM 的 UTF-8 字符串。

在 Windows 平台，可以同样运行：

```
zabbix_sender.exe [options]
```

自 Zabbix 1.8.4 开始，Zabbix_sender 实时发送方案已得到改进，收集传递临近连续的多个值之后，将它们合并，通过同一个请求发送到服务器。放在同一个堆栈的时间可以不超过 0.2 秒，但最大 pooling 时间仍然为 1 秒。



如果配置文件中的参数实体无效（不按 `parameter=value` 格式），则 Zabbix sender 就会终止。

7. get

Zabbix get 用于连接 Zabbix agent 并从 agent 上检索需要的信息。

该工具常用于对 Zabbix agent 进行故障检修。

运行 Zabbix get

在 UNIX 平台上运行 Zabbix get 以获得指定的 agent 上的进程负载：

```
shell> cd bin
shell> ./zabbix_get -s 127.0.0.1 -p 10050 -k system.cpu.load[all,avg1]
```

在 Web 网站上运行 Zabbix get 来获取字符串：

```
shell> cd bin
shell> ./zabbix_get -s 192.168.1.1 -p 10050 -k "web.page.regexp
[www.zabbix.com,,, \"USA: ([a-zA-Z0-9.-]+)\",, \1]"
```

Zabbix get 支持的命令行参数如表 6-1 所示。

表 6-1 Zabbix get 支持的命令行参数

<code>-s --host <主机名或 IP></code>	指定 agent 所在的 host 名或 IP 地址
<code>-p --port <端口号></code>	指定 host 所运行的 agent 的端口号，默认为 10050
<code>-I --source-address <IP 地址></code>	指定源 IP 地址
<code>-k --key <item key></code>	指定需要检索的 item 键名
<code>-h --help</code>	显示本帮助
<code>-V --version</code>	显示版本号

在 Windows 平台下运行 Zabbix get 类似：

```
zabbix_get.exe [options]
```

其他有关 Zabbix get 的详情，可自行参阅 https://www.zabbix.com/documentation/3.2/manpages/zabbix_get。

6.4 Consul

Consul 是 HashiCorp 公司推出的开源工具，用于实现分布式系统的服务发现、监控与配置。与其他分布式服务注册与发现的方案相比，Consul 的方案更“一站式”，内置了服务注册与发现框架、分布一致性协议实现、健康检测、Key/Value 存储、多数据中心方案等，而且不再需要依赖其他工具（比如 ZooKeeper 等），使用起来也较为简单。Consul 是用 Go 语言开发的，因





此具有天然可移植性（支持 Linux、Mac OS X、FreeBSD、Solaris 和 Windows）；安装包仅包含一个可执行文件，方便部署，与 Docker 等轻量级容器可无缝配合。

6.4.1 Consul 架构

Consul 是一个复杂的系统，该系统由许多不同的部件组成。下面介绍 Consul 的系统架构。

1. 术语

在描述体系结构之前，首先介绍该系统常用的几个术语。

- **Agent**——agent 是长期运行在每个 Consul 集群成员节点上的守护进程。通过 `consul agent` 命令启动。agent 有 client 和 server 两种模式。由于每个节点都必须运行 agent，所以节点要么是 client 要么是 server。所有的 agent 都可以调用 DNS 或 HTTP API，并负责检查和维护服务同步。
- **Client**——运行 client 模式的 agent，将所有的 RPC 转发到 server。client 是相对无状态的。client 唯一所做的是在后台参与 LAN gossip pool。只消耗少量的资源和少量的网络带宽。
- **Server**——运行 server 模式的 agent，参与 Raft quorum，维护集群的状态，响应 RPC 查询，与其他数据中心交互 WAN gossip，转发查询到 leader 或远程数据中心。
- **Datacenter**——数据中心的定义是显而易见的，有一些细节是必须考虑的。例如，在 EC2，多个可用性区域是否被认为组成了单一的数据中心？我们定义数据中心在同一个网络环境中——私有的、低延迟、高带宽。这包括基于公共互联网的环境，但是对于我们而言，在同一个 EC2 中的多个可用性区域会被认为是在同一个数据中心中。
- **Consensus**——consensus 意味着 leader election 机制，以及事务的顺序。由于这些事务基于一个有限状态机，consensus 的定义意味着复制状态机的一致性。
- **Gossip**——Consul 建立在 Serf 之上，提供了完整的 Gossip 的协议，用于成员维护故障检测、事件广播。
- **LAN Gossip**——指的是 LAN gossip pool，包含位于同一个局域网或者数据中心的节点。
- **WAN Gossip**——指的是 WAN gossip pool，只包含 server 节点，这些 server 主要分布在不同的数据中心，或者通信是基于互联网或广域网的。
- **RPC**——远程过程调用。它是允许 client 请求 server 的请求/响应机制。

2. 架构

Consul 架构图可参见官方文档。





首先，Consul 支持多数据中心（datacenter），从这个架构图中可以看到有两个数据中心，分别标记为“datacenter 1”和“datacenter 2”。

每个数据中心都是由 server 和 client 组成的。基于故障处理和性能的平衡的考虑，建议有 3~5 个 server。随着机器的增多，consensus 会越来越慢。client 没有限制，可以很容易地扩展到成千上万。

同一个数据中心的所有节点都要加入 Gossip 协议。这意味着 Gossip pool 包含给定数据中心的所有节点。出于以下目的：

- 没有必要为 client 配置 server、地址参数；发现是自动完成的。
- 节点故障检测的工作不是放置在 server 上，而是分布式的。这使故障检测比心跳机制更具可扩展性。
- 当重要的事件发生时，如 leader election（领导人选举），可用来作为消息层的通知。

每个数据中心的 server 都属于一个 Raft 对等端，这意味着它们一起工作。在它们的中间选出一个 leader，leader 是有额外职责的，需要负责处理所有的查询和事务。事务也必须通过 consensus 协议复制到所有的伙伴中。由于这一要求，当非 leader server 接收到一个 RPC 请求时，会转发到集群的 leader 中。

server 节点也作为 WAN gossip pool 的一部分。这个 pool 与 LAN pool 是不同的，它为具有更高延迟的网络响应做了优化，并且可能包含其他 Consul 集群的 server 节点。设计 WAN gossip pool 的目的是让数据中心能够以 low-touch（低接触）的方式发现彼此。将一个新的数据中心加入现有的 WAN Gossip 是很容易的。因为 pool 中的所有 server 都是可控制的，所以能够满足跨数据中心的要求。当一个 server 接收到不同的数据中心的要求时，它把这个请求转发给相应数据中心的任一 server。然后，接收到请求的 server 可能会转发给本地 leader。

多个数据中心之间是低耦合的，但由于需要满足故障检测、连接缓存和多路复用等需求，跨数据中心被设计为能够快速和可靠地响应请求。

6.4.2 Consul agent

Consul agent 是 Consul 的核心过程。agent 的作用包括维护成员信息、注册服务、运行检查、响应查询等。agent 必须在 Consul 集群的每个节点上运行。

agent 可以以 client 或 server 模式运行。server 模式运行的节点用于维护 Consul 集群的状态，提供故障情况下的强一致性和可用性。client 模式则相对轻量很多，只需维护自身的状态。





1. 运行和停止 agent

使用 `consul agent` 命令运行 agent:

```
$ consul agent -data-dir=/tmp/consul
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
    Node name: 'Armons-MacBook-Air'
    Datacenter: 'dc1'
    Server: false (bootstrap: false)
    Client Addr: 127.0.0.1 (HTTP: 8500, DNS: 8600, RPC: 8400)
    Cluster Addr: 192.168.1.43 (LAN: 8301, WAN: 8302)
    Atlas: (Infrastructure: 'hashicorp/test' Join: true)

==> Log data will now stream in as it occurs:
```

```
[INFO] serf: EventMemberJoin: Armons-MacBook-Air.local 192.168.1.43
...
```

上面的输出内容反映了几个重要信息。

- **Node name (节点名称)**: agent 的唯一名称。默认是主机名称, 但可以使用 `-node` 标识来自定义。
- **Datacenter (数据中心)**: agent 所运行的 datacenter。Consul 支持多 datacenter, 但为了工作更加有效, 每个节点必须配置其 datacenter。可以通过 `-datacenter` 标识来设置 datacenter。对于单 datacenter 的配置, agent 默认配置到“dc1”。
- **Server**: 指明 agent 运行在 server 还是 client 模式。server 模式的节点需要存储集群状态和处理查询, 比 client 模式要承担额外负担。此外, 一个 server 可以是在“bootstrap”模式。多个 server 不能在“bootstrap”模式, 因为这将使集群中出现不一致的状态。
- **Client Addr**: client 提供给 agent 的接口的地址, 包含 HTTP、DNS 和 RPC 接口的端口号。可以使用 `-rpc-addr` 标识来指定地址或端口。
- **Cluster Addr**: 该地址及端口集合用于集群内的 Consul agent 之间的通信。不是所有的 Consul agent 都使用相同的端口号, 但地址必须是所有其他节点能够访问的。
- **Atlas**: 显示节点所注册的 Atlas。同时指明了 `auto-join` (自动加入) 是否启用。可以使用 `-atlas` 来设置 Atlas, 使用 `-atlas-join` 来启用 auto-join。

使用 `Ctrl-C` 或杀掉 agent 进程的方式来停止 agent。





2. agent 的生命周期

集群中的每个 agent 都会经历一个生命周期。理解这个生命周期,可以更好地建立一个 agent 之间的交互,以及了解集群是如何管理节点的。

当一个 agent 第一次启动时,它不知道集群中的任何其他节点。要发现其他节点,agent 就必须加入(join)集群,可以使用 join 命令进行配置。一旦一个节点加入,此信息就会传播到整个集群,这意味着所有节点最终会知道对方。如果 agent 是一个 server,则现有的 server 将开始复制到信息新节点。

在网络故障的情况下,一些节点可能无法被其他节点访问。在这种情况下,无法访问的节点被标记为 failed(失败)。网络故障和 agent 崩溃这两种情况是没办法区分的,因此这两种情况下的处理方式相同。一旦某节点被标记为 failed,该信息就会在服务目录中进行更新。

当一个节点主动离开(leaves)时,会被集群标志为 left(离开)。与 failed 不同的是,所有由节点提供的服务会被注销。如果 agent 是一个 server,则复制将会停止。

为了防止死亡节点(在 failed 或 left 状态的节点)的积累,Consul 将自动删除目录中的死亡节点。这个过程被称为 reaping(收获)。reaping 的时间间隔默认是 72 小时,其值是可配置的。

3. DNS 接口

DNS 接口是 Consul 的主要查询接口之一,允许应用进行服务发现。

DNS 接口常用的配置选项有 client_addr、ports.dns、recursors、domain、dns_config 等。默认 Consul 会监听 127.0.0.1:8600 的 DNS 查询接口。

可以使用 dig 来查询 Consul 的 DNS server:

```
$ dig @127.0.0.1 -p 8600 redis.service.dcl.consul. ANY
```

节点查找

节点查找的格式如下:

```
<node>.node[.datacenter].<domain>
```

完整的示例如下:

```
$ dig @127.0.0.1 -p 8600 foo.node.consul ANY
```

```
; <<>> DiG 9.8.3-P1 <<>> @127.0.0.1 -p 8600 foo.node.consul ANY
; (1 server found)
;; global options: +cmd
;; Got answer:
```





```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 24355
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;foo.node.consul.      IN ANY

;; ANSWER SECTION:
foo.node.consul.      0   IN A   10.1.10.12

;; AUTHORITY SECTION:
consul.      0   IN SOA ns.consul. postmaster.consul. 1392836399 3600 600 86400 0
```

服务查找

服务查找支持两种方式：标准（standard）查找和严格的 RFC 2782 查找。

（1）标准（standard）查找。

标准服务查找的格式如下：

```
[tag.]<service>.service[.datacenter].<domain>
```

完整的示例如下：

```
$ dig @127.0.0.1 -p 8600 consul.service.consul SRV

; <<>> DiG 9.8.3-P1 <<>> @127.0.0.1 -p 8600 consul.service.consul ANY
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 50483
;; flags: qr aa rd; QUERY: 1, ANSWER: 3, AUTHORITY: 1, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;consul.service.consul.      IN SRV

;; ANSWER SECTION:
consul.service.consul.      0   IN SRV 1 1 8300 foobar.node.dc1.consul.

;; ADDITIONAL SECTION:
```





```
foobar.node.dcl.consul. 0 IN A 10.1.10.12
```

(2) 严格的 RFC 2782 查找。

严格的 RFC 2782 查找的格式如下：

```
_<service>._<protocol>.service[.datacenter][.domain]
```

完整的示例如下：

```
$ dig @127.0.0.1 -p 8600 _rabbitmq._amqp.service.consul SRV

; <<>> DiG 9.8.3-P1 <<>> @127.0.0.1 -p 8600 _rabbitmq._amqp.service.consul ANY
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<- opcode: QUERY, status: NOERROR, id: 52838
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;_rabbitmq._amqp.service.consul. IN SRV

;; ANSWER SECTION:
_rabbitmq._amqp.service.consul. 0 IN SRV 1 1 5672 rabbitmq.node1.dcl.
consul.

;; ADDITIONAL SECTION:
rabbitmq.node1.dcl.consul. 0 IN A 10.1.11.20
```

有关 RFC 2782 协议，读者可以自行参阅 <https://tools.ietf.org/html/rfc2782>。

预备查询 (Prepared Query) 的查找

预备查询查找的格式如下：

```
<query or name>.query[.datacenter].<domain>
```

其中 `datacenter` 是可选的选项，是 Consul agent 的 `datacenter`。`query or name` 是现有预备查询的 ID 或给定名字。

4. HTTP API

Consul 主要的接口是 RESTful HTTP API。该 API 可以用来对节点、服务、检测、配置等执



行 CRUD 操作。端点进行了版本控制，以便向后兼容变化。

每个端点管理 Consul 的不同方面：

- acl——访问控制列表；
- agent——Consul agent；
- catalog——节点和服务；
- coordinate——网络协调；
- event——用户事件；
- health——健康检测；
- kv——Key/Value 存储；
- operator——Consul 操作工具；
- query——预备查询；
- session——会话；
- status——Consul 系统状态。

5. 配置

agent 可以通过命令行或配置文件来指定不同的配置选项。所有的配置选项完全是可选的。

当加载配置时，Consul 以词法顺序从文件和目录中加载配置。例如，配置文件 `basic_config.json` 将在 `extra_config.json` 前被处理。后来指定的配置将被合并到先前指定的配置中。在大多数情况下，“合并”指的是较新版本覆盖较早的版本。在某些情况下，如事件处理程序，合并会追加处理程序到现有的配置中。

Consul 也支持当它接收到 SIGHUP 信号时重配置。

下面是配置文件的示例：

```
{
  "datacenter": "east-aws",
  "data_dir": "/opt/consul",
  "log_level": "INFO",
  "node_name": "foobar",
  "server": true,
  "watches": [
    {
      "type": "checks",
      "handler": "/usr/bin/health-check-handler.sh"
```



```

    }
  ],
  "telemetry": {
    "statsite_address": "127.0.0.1:2180"
  }
}

```

以下是支持 TLS 的配置文件的示例：

```

{
  "datacenter": "east-aws",
  "data_dir": "/opt/consul",
  "log_level": "INFO",
  "node_name": "foobar",
  "server": true,
  "addresses": {
    "https": "0.0.0.0"
  },
  "ports": {
    "https": 8080
  },
  "key_file": "/etc/pki/tls/private/my.key",
  "cert_file": "/etc/pki/tls/certs/my.crt",
  "ca_file": "/etc/pki/tls/certs/ca-bundle.crt"
}

```

6. 服务定义

服务定义的示例如下：

```

{
  "service": {
    "name": "redis",
    "tags": ["master"],
    "address": "127.0.0.1",
    "port": 8000,
    "enableTagOverride": false,
    "checks": [
      {
        "script": "/usr/local/bin/check_redis.py",
        "interval": "10s"
      }
    ]
  }
}

```



```
    }  
  ]  
}  
}
```

多服务定义的示例如下：

```
{  
  "services": [  
    {  
      "id": "red0",  
      "name": "redis",  
      "tags": [  
        "master"  
      ],  
      "address": "127.0.0.1",  
      "port": 6000,  
      "checks": [  
        {  
          "script": "/bin/check_redis -p 6000",  
          "interval": "5s",  
          "ttl": "20s"  
        }  
      ],  
    },  
    {  
      "id": "red1",  
      "name": "redis",  
      "tags": [  
        "delayed",  
        "slave"  
      ],  
      "address": "127.0.0.1",  
      "port": 7000,  
      "checks": [  
        {  
          "script": "/bin/check_redis -p 7000",  
          "interval": "30s",  
          "ttl": "60s"  
        }  
      ]  
    }  
  ]  
}
```



```
    }  
  ]  
},  
...  
]  
}
```

7. 检测 (Check) 定义

脚本检测定义的示例如下:

```
{  
  "check": {  
    "id": "mem-util",  
    "name": "Memory utilization",  
    "script": "/usr/local/bin/check_mem.py",  
    "interval": "10s",  
    "timeout": "1s"  
  }  
}
```

HTTP 检测定义的示例如下:

```
{  
  "check": {  
    "id": "api",  
    "name": "HTTP API on port 5000",  
    "http": "http://localhost:5000/health",  
    "interval": "10s",  
    "timeout": "1s"  
  }  
}
```

TCP 检测定义的示例如下:

```
{  
  "check": {  
    "id": "ssh",  
    "name": "SSH TCP on port 22",  
    "tcp": "localhost:22",  
    "interval": "10s",
```

```
    "timeout": "1s"
  }
}
```

TTL 检测定义的示例如下：

```
{
  "check": {
    "id": "web-app",
    "name": "Web App Status",
    "notes": "Web app does a curl internally every 10 seconds",
    "ttl": "30s"
  }
}
```

Docker 检测定义的示例如下：

```
{
  "check": {
    "id": "mem-util",
    "name": "Memory utilization",
    "docker_container_id": "f972c95ebf0e",
    "shell": "/bin/bash",
    "script": "/usr/local/bin/check_mem.py",
    "interval": "10s"
  }
}
```

多检测定义的示例如下：

```
{
  "checks": [
    {
      "id": "chk1",
      "name": "mem",
      "script": "/bin/check_mem",
      "interval": "5s"
    },
    {
      "id": "chk2",
      "name": "/health",

```

```

    "http": "http://localhost:5000/health",
    "interval": "15s"
  },
  {
    "id": "chk3",
    "name": "cpu",
    "script": "/bin/check_cpu",
    "interval": "10s"
  },
  ...
]
}

```

6.5 ZooKeeper

ZooKeeper 分布式服务框架是 Apache Hadoop 的一个子项目，主要用来解决分布式应用中经常遇到的一些数据管理问题，如统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。ZooKeeper 的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

6.5.1 ZooKeeper 简介

正如 ZooKeeper（动物园管理员）其名称的含义，整个 Coordinating Distributed Systems（协调分布式系统）就是一个 Zoo（动物园），系统里面的各种组件被定义为各种动物（比如，Hadoop 是大象，Whirr 就是飞鸟，Hive 是蜜蜂，Pig 是肥猪），而 ZooKeeper 在里面起协调管理作用。

ZooKeeper 是分布式应用的高性能协调服务。它暴露了常见的服务——例如，命名、配置管理、同步和组服务——在一个简单的界面，让你不必从头开始编写它们。它被设计为易于编程，使用与文件系统目录树结构类似的数据模型。ZooKeeper 使用 Java 编写。

协调服务（coordination services）的开发是非常困难的。它们特别容易出错，比如会出现竞态条件和死锁。ZooKeeper 的出现，正好缓解了分布式应用程序从头开始实施协调服务所产生的问题。

1. 设计目标

ZooKeeper 的设计目标：

- 操作简单——ZooKeeper 主要用来协调处理分布式任务（通过一个叫多层次的命名空间，

这种命名空间很类似文件系统）。一个命名空间就是一个数据寄存器，被称为 `znode`，按照 ZooKeeper 的说法，多层次的命名空间就是文件与目录的关系。但 ZooKeeper 的数据保存在内存中，可以实现高吞吐量和低延迟。

- 自我复制——像图 6-7 所示的这个分布式系统，ZooKeeper 主要在各个 Server 间复制数据，ZooKeeper 服务必须彼此知道对方的存在。它们维持一个内存中的状态图像，以及持久存储中的事务日志和快照。只要大多数的 ZooKeeper 机器可以运行，ZooKeeper 就可以提供正常的服务。当一个 Client 需要的服务可通过 TCP 连接到一个 Server 时，如果这个 Server 挂掉了，则它就会自动连接另一个 Server。

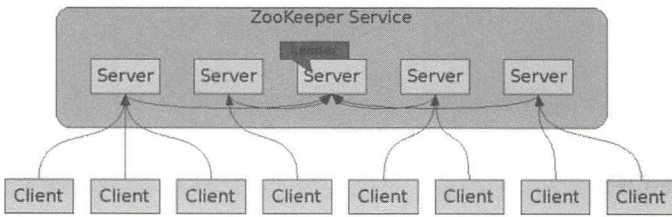


图 6-7 ZooKeeper 系统示意图

- 有序：ZooKeeper 通过更新一个计数器来反映 ZooKeeper 的事务顺序，子操作可以通过这个计数器来实现更高层次的抽象，例如原语同步。
- 快速：ZooKeeper 尤其在读取时表现的性能更为强悍，因为 ZooKeeper service 可以同时由很多机器提供，而且读的速度是写的速度的 10 倍。

2. 数据模型和多层次命名空间

ZooKeeper 的多层次命名空间就像常见的文件系统，每个节点的路径都是唯一的，如图 6-8 所示。

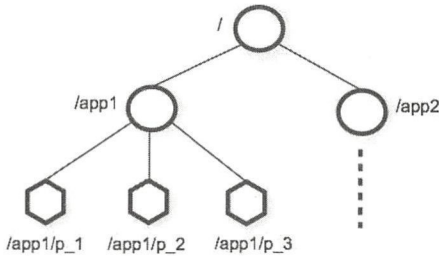


图 6-8 ZooKeeper 命名空间

3. ZooKeeper 节点和临时节点

ZooKeeper 的节点是通过像树一样的结构来进行维护的，并且每一个节点通过路径来标示

和访问。除此之外，每一个节点还拥有自身的一些信息，包括数据、数据长度、创建时间、修改时间，等等。从这样一类既含有数据，又作为路径表标示的节点的特点中可以看出，ZooKeeper 的节点既可以被看作文件，又可以被看作目录，它同时具有二者的特点。为了便于表达，一般使用 `znode` 来表示所讨论的 ZooKeeper 节点。

具体地说，`znode` 通过管理包含数据、ACL（access control list，访问控制列表）、时间戳的版本号数据结构，来实现缓存生效及协调更新。每当 `znode` 中的数据更新后，它所维护的版本号将增加，这非常类似于数据库中计数器时间戳的操作方式。

另外，`znode` 具有原子性操作的特点：在命名空间中，每一个 `znode` 的数据将被原子地读写。读操作将读取与 `znode` 相关的所有数据，写操作将替换所有的数据。除此之外，每一个节点都有一个访问控制列表，这个访问控制列表规定了用户操作的权限。

ZooKeeper 中同样存在临时节点。这些节点与 `session` 同时存在，当 `session` 生命周期结束时，这些临时节点也将被删除。临时节点在某些场合也发挥着非常重要的作用。

4. 有条件的 update 和 watch

ZooKeeper 支持 `watch` 的概念。`client` 可以在 `znode` 上设置 `watch`。当 `znode` 变更时，`watch` 将被触发并移除。当 `watch` 被触发后，`client` 就会收到一个数据包，说明 `znode` 已经改变了。如果 `client` 和 ZooKeeper server 之间的连接被断开，则 `client` 将收到本地通知。

5. 保证

ZooKeeper 的速度非常快，非常简单。为了支撑其更复杂的服务，提供了如下的保证：

- Sequential Consistency（顺序一致性）——`client` 的更新将在它们应用时按照顺序进行发送。
- Atomicity（原子性）——更新非成功即失败。
- Single System Image（单一系统映像）——一个 `client` 将看到相同的服务视图（view），而不管它连接哪个服务器。
- Reliability（可靠性）——一旦更新已被应用，它会从那个时候起一直持续，直到 `client` 再次更新为止。
- Timeliness（时效性）——该系统的 `client` 视图保证在一定时间内是最新的。

6. API 简单

ZooKeeper 提供了非常简单的编程接口，它仅支持以下这些操作：

- `create`——在树中的位置创建一个节点；
- `delete`——删除一个节点；

- exists——测试一个节点是否出现在某个位置；
- get data——从一个节点读取数据；
- set data——写入数据到节点；
- get children——检索节点的子节点的列表；
- sync——等待数据被传播。

7. 实现

图 6-9 展示了 ZooKeeper 服务的高层组件。除了请求处理器（request processor），构成 ZooKeeper 服务的每个服务器都有一个备份。

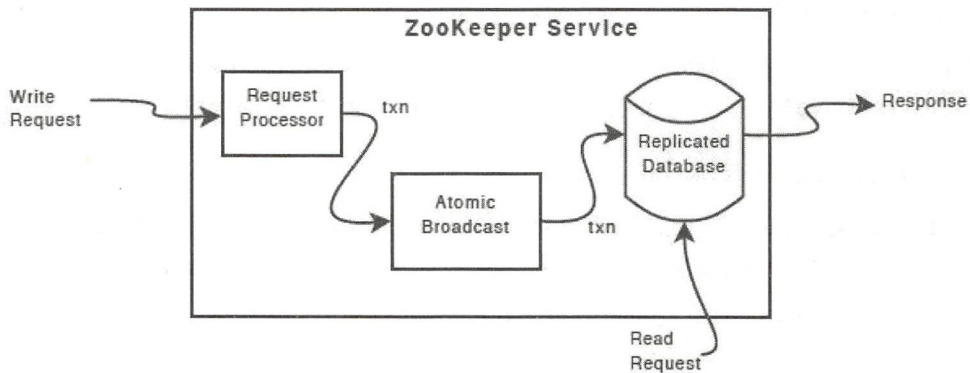


图 6-9 ZooKeeper 服务的高层组件

复制的数据库（replicated database）是一个内存数据库，包含整个数据树。为了可恢复，更新日志会被记录到磁盘，并且在更新这个内存数据库之前，先序列化到磁盘。

每个 ZooKeeper server 都为 client 提供服务。client 只连接一个 server，并提交请求。读请求直接由本地的复制数据库提供数据。对服务状态进行修改的请求、写请求通过一个约定的协议进行通信。

作为这个协议的一部分，所有的写请求都被传送到一个叫“leader”的 server，而其他的 server 叫作“follower”。follower 从 leader 接收信息修改的提议，并决定是否同意进行修改。当 leader 发生故障时，协议的信息层（messaging layer）关注 leader 的替换，并同步到所有的 follower。

ZooKeeper 采用一个自定义的信息原子操作协议，由于信息层的操作是原子性的，ZooKeeper 能保证本地的复制数据库不会产生不一致。当 leader 收到一个写请求时，它计算出写之后系统的状态，把它变成一个事务。

6.5.2 ZooKeeper 内部工作原理

1. 原子广播

ZooKeeper 的核心就是消息处理的原子性，能够保持所有的 server 同步。

2. 保证、属性和一些定义

ZooKeeper 能够保证消息处理原子性的特性包括：

- 可靠的消息传递——如果一个消息 m 被某个 server 接收了，那么基本上所有 server 肯定也都收到了该消息。
- 顺序接收——如果消息 a 先于消息 b 被某个 server 接收，那么所有 server 接收 a 都会先于 b 。 a 和 b 同时传递消息的话，要么 a 在前，要么 b 在前，不会出现并行或混乱冲突的情况。
- 因果关系——如果消息 a 先于 b ， b 又先于 c ，那么 a 肯定是先于 c 的。

ZooKeeper 消息系统被设计得高效、可靠，实现和维护都很简单。

由于我们需要大量使用消息，所以需要 ZooKeeper 平均每秒能够处理成千上万的请求。

尽管我们使用 $k+1$ 个正常运行的 server 收发消息，但还是必须能够恢复比如断电导致的所有 server 停止工作的情况（相对单个 server 出问题的情况）。

ZooKeeper 的协议假设我们能够在点对点的 server 中构造 FIFO 消息通道。一般相类似的服务总是假设消息会丢失或重复，我们会假设 FIFO 通道是可靠的。由于使用 TCP 连接，基于 TCP 连接具有以下特点：

- 消息顺序传递——消息 m 总是会在所有之前的消息之后传递。因此，如果消息 m 丢失了，那么 m 之后的消息也都会丢失。
- FIFO 管道关闭后就接收不到消息了——如果 FIFO 消息管道关闭了，则不可能从该管道中接收消息。

FLP 证明如果发生了错误，则一致性就不可能在分布环境中实现。为了在出错的时候实现一致性，使用 timeout 机制来实现。

使用 timeout 机制是为了证明 server 的存活，而不是证明 server 的正确性。当 timeout 机制停止工作（计时发生故障）时，消息系统会挂起，但是依然能够保证一致性正常工作。

在描述 ZooKeeper 消息协议时，经常会谈到如下概念：

- 数据包——通过 FIFO 通道发送的一系列字节流。
- 提议（Proposal）——一个协议单元，提议通过 ZooKeeper server 的 quorum 交换数据包

来表决。大多数提议包含消息。但是有个特别的就是 NEW_LEADER 协议是不带消息的。

- 消息——字节流会自动广播到其他 ZooKeeper server。提议和同意提议在传递的时候都会附带消息。

就如以上提到的，ZooKeeper 保证所有消息的顺序一致，也保证所有提议的顺序。ZooKeeper 使用 ZooKeeper 事务 id (zxid) 来保证提议的顺序。所有的提议都会被加上一个 zxid，当这个提议被发起后，通过 zxid 就能反映提议的顺序。提议被发送到所有的 ZooKeeper server，然后其中一个 server 如果认可该提议，则这个 server 就会提交这个提议。如果提议包含一条消息，当提交提议的时候这个消息也会一起被提交。

确认该协议意味着 server 持久化存储这个提议。quorum 要求任何一个 quorum 必须至少有一个 server。至少一半以上的 server 同意该提议，该提议才有效。

zxid 包含两个部分：epoch 和计数器。zxid 用一个 64bit 的数字实现，高 32 位表示 epoch，低 32 位表示计数器。因为 zxid 的两部分都是用数字表示的，数字改变就代表 leader 的改变。每次产生一个新的 leader，就有一个数字特定地表示这个新的 leader。

因为和其他 leader 的提议肯定不同，所以保证了提议的唯一性。

ZooKeeper 消息系统由两部分组成：

- leader 激活——这个阶段需要选举一个 leader，然后建立正确的系统状态，准备好接受提议。
- 消息传递——这个阶段 leader 接受提议，而且协调提议的正确传递。

3. leader 激活

leader 激活包括 leader election。当前 ZooKeeper 中有两个 leader election 算法：LeaderElection 和 FastLeaderElection（AuthFastLeaderElection 通过 UDP 通信，而且允许各个 server 使用一组简单的认证方式来避免 IP 欺骗）。ZooKeeper 消息并不关心使用哪一种具体选举法。只要选举结果满足以下要求就好：

- leader 的 zxid 必须是 follower 中最高的；
- 将法定数量的 server 提交给 leader。

leader 激活过程只有以下几步操作：

- follower 在和 leader 同步过后，会确认收到一个 NEW_LEADER 提议；
- follower 只会收到一个单独 server 的特定的 zxid 的 NEW_LEADER 提议时才会确认；
- 当法定数量的 follower 都确认时，新的 leader 将提交这个 NEW_LEADER 提议；
- 当 NEW_LEADER 提议提交后，follower 就会提交所有的接收自 leader 的状态；
- 这个 leader 必须在 NEW_LEADER 提议被提交通过后，才能接收其他新的提议。

如果 leader election 过程意外结束了，因为 NEW_LEADER 提议还没有被提交通过，所以这个 leader 没有任何选票，不会出任何问题。当意外发生了，当前 leader 和其他的 follower 都会因为连不上而 timeout，然后会重新开始新的 leader election。

4. 激活消息

leader 激活是最烦琐的。一旦一个 leader 被确定了，它就开始接受提议。只要这个 leader 还在，就不会产生其他的 leader，因为其他 leader 得不到任何选票，也就不会被选举成为 leader。如果一个新的 leader 产生，那么旧的 leader 肯定联系不上了。新 leader 会清理旧 leader 的所有烂摊子。

ZooKeeper 的消息处理方式和经典的双向提交确认很像，如图 6-10 所示。

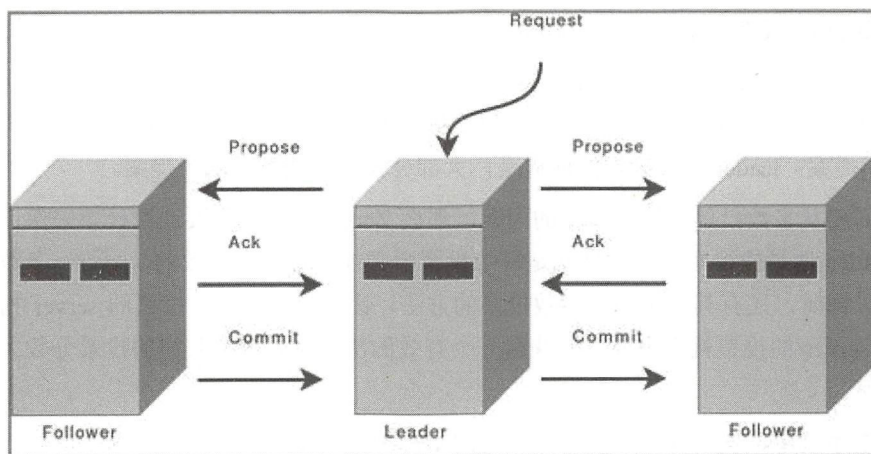


图 6-10 ZooKeeper 的消息处理方式

所有的联系通道都是 FIFO，所以所有处理都是有顺序的。有以下的操作限制：

- leader 发送提议给所有 server 是挨个发送的。因此，每个 server 接收到请求也是依序接收到的。因为 FIFO 的特性决定了 server 必须是依序收到的。
- server 顺序地处理收到的消息，这就意味着每个消息都必须按顺序确认，而且 leader 也是按顺序收到确认的消息。由于 FIFO 的特性，如果消息 m 被写入持久化存储，那么在 m 被写入之前，被提议的消息也都被写入持久化存储。
- 一旦法定人数的 follower 同意这个提议，leader 会发布一个 COMMIT 消息给所有的 server。由于消息已经被一个一个地确认了，COMMIT 消息会一个一个地发送给 server，每个 server 也会都接收到。
- COMMIT 消息会被 server 按顺序处理，每个 server 会在该提议提交的时候一起传递消息。

5. 与 Paxos 算法比较

这个是不是很像 Multi-Paxos 算法呢？并不是。Multi-Paxos 算法要求某种算法假设只有一个 leader，我们不能依赖这种假设。相反我们使用 leader 激活过程去替换 leader，或者使用旧的 leader 确认它还是有效的。

那么这是不是就是 Paxos 算法呢？激活消息的阶段是不是很像 Paxos 算法的阶段二？实际上，消息激活看起来就像 Paxos 算法的无须处理失败情况的第二个阶段提交。激活消息与两者都不同，因为它具有交叉提议排序要求（cross proposal ordering requirements）。如果所有的数据包不维护严格的 FIFO 顺序，算法就会失败，不可靠。leader 激活阶段也和这两种算法不同。实际上，使用 epoch 的方法就可以跳过未提交的提议，而且不必担心一个 zxid 会有多个提议。

6. quorum（投票）机制

quorum 保证了自动广播和 leader election 的系统一致性。默认的，ZooKeeper 采用 majority quorum（多数派投票机制），这就意味着每次提议的投票必须有多个 server 通过。典型的就是 leader 选举提案：leader 会被确定的前提是，大部分投票都认可了这个提案。

如果需要从多数投票中提取重要的因素，那么 ZooKeeper 只需要通过投票来保证某个提议（比方 leader 选举提议）的有效性就是每个投票中必须包含一个有效的 server，多数投票保证这个因素。同时，还有其他不同于多数投票的方法，比如可以对每个投票的 server 指定权重，这样某些 server 的投票就更重要。要获得一个有效的决议，只需要获得的投票分数大于总投票的分数即可。

在分层系统中，使用权重加权构造系统的结构被广泛使用。在这种情况下，一般将所有的 server 分成几个组，然后给不同的组指定不同的权重。要形成决议，必须从大组 G 中得到足够多的 server 的支持，这样只要从小组 g 中获得选票分数大于 g 的选票分数总和即可。有趣的是，这种结构允许更小的投票确定一个提议。如果有 9 个 server，则分成 3 组，然后每组指定权重为 1，这样就可以在只得到 4 票分数的情况下确定该提议有效了。具体就是有两组 server 中各自有 2 票同意。这种情况是有效的，某个小组中的大部分成员同意了，就表示在这个小组同意了。

在 ZooKeeper 中提供了接口，用于配置 ZooKeeper 工作在多数投票、权重加权，或者分组结构的模式下。

7. 日志

ZooKeeper 使用 slf4j 作为日志的抽象层，默认使用 log4j 1.2 来做实际的日志工作。为了更好地支持嵌入，在未来的计划中，会将选择最后日志实现的权利交给最终用户。因此，一定要使用 slf4j api 在代码中编写日志语句，但可以在运行时配置 log4j 来指明如何记录日志。需要注意的是，slf4j 没有 FATAL 级别的日志，之前 FATAL 级别的消息已被移至 ERROR 级别。

slf4j 主要有以下几个日志级别：ERROR、WARN、INFO、DEBUG、TRACE。

在生产环境一般使用 INFO 级别。

以下是 slf4j 的用法示例。

静态消息日志：

```
LOG.debug("process completed successfully!");
```

需要创建参数化的消息时，使用格式锚：

```
LOG.debug("got {} messages in {} minutes", new Object[]{count, time});
```

Logger 需要在它们所使用的类中命名：

```
public class Foo {  
    private static final Logger LOG = LoggerFactory.getLogger(Foo.class);  
    ....  
    public Foo() {  
        LOG.info("constructing Foo");  
    }  
}
```

异常处理：

```
try {  
    // ...  
} catch (XYZException e) {  
    // ...  
    LOG.error("Something bad happened", e);  
    // ...  
}
```

6.5.3 例子：ZooKeeper 实现 barrier 和 producer-consumer queue

本节将演示如何使用 ZooKeeper 实现两种常用的分布式数据结构：barrier（栅栏）和 producer-consumer queue（生产者-消费者队列）。我们为此还分别实现了两个类：Barrier 和 Queue。文中的例子假设已经成功运行了 ZooKeeper 服务器。

barrier 是这样一个类：它会阻塞所有节点上的等待进程，直到某一个被满足，然后所有的节点继续进行。

上述两种最基本的类都使用了下面公用的代码：

```
static ZooKeeper zk = null;  
static Integer mutex;
```

```
String root;

SyncPrimitive(String address) {
    if(zk == null){
        try {
            System.out.println("Starting ZK:");
            zk = new ZooKeeper(address, 3000, this);
            mutex = new Integer(-1);
            System.out.println("Finished starting ZK: " + zk);
        } catch (IOException e) {
            System.out.println(e.toString());
            zk = null;
        }
    }
}

synchronized public void process(WatchedEvent event) {
    synchronized (mutex) {
        mutex.notify();
    }
}
```

两个类都扩展了 `SyncPrimitive`。用这种方式，执行步骤和 `SyncPrimitive` 的构造函数的所有原语差不多。为了保持例子简单，我们创建一个 `ZooKeeper` 对象，在每一次实例化 `barrier` 对象或 `queue` 对象时，我们声明一个静态的变量来引用这个对象。随后的 `Barrier` 实例和 `Queue` 实例检查一个 `ZooKeeper` 对象是否存在。或者，我们也可以让应用程序创建一个 `ZooKeeper` 对象，并将其传递给 `Barrier` 和 `Queue` 的构造函数。

使用 `process()` 方法来处理 `watch` 的通知。将下面讨论如何设置 `watch` 的代码。一个 `watch` 是一个内部的数据结构，它能够使 `ZooKeeper` 通知节点的改变。例如，如果一个 `client` 正在等待其他 `client` 离开一个 `Barrier`，那么它可以给一个特定的节点设置一个 `watch` 并且等待修改。

1. Barrier

`Barrier` 使一组进程可以同步一个计算的开始和结束过程。这种实现的总体思想是有一个 `Barrier` 节点作为每一个进程节点的父节点。假如这个 `Barrier` 节点为“/b1”，则每一个进程“p”创建一个节点“/b1/p”。一旦有足够的进程创建了它们对应的节点，加入的进程就可以开始计算。

在这个例子中，每一个进程代表一个 Barrier 对象，并且它的构造函数有这些参数：

- ZooKeeper server 的地址（例如，“zoo1.foo.com:2181”）；
- ZooKeeper 中 Barrier 节点的路径（例如，“/b1”）；
- 这组进程的数量。

Barrier 的构造函数传递 Zookeeper server 的地址给父类的构造器。如果之前没有这个实例，则父类会创建一个 ZooKeeper 实例。Barrier 的构造函数在 ZooKeeper 上创建一个节点，这个节点是所有进程节点的父节点，我们称它为 root（注意，它不是 ZooKeeper 的根“/”）。

```
/**
 * Barrier 构造函数
 *
 * @param address
 * @param root
 * @param size
 */
Barrier(String address, String root, int size) {
    super(address);
    this.root = root;
    this.size = size;

    // 创建 Barrier 节点
    if (zk != null) {
        try {
            Stat s = zk.exists(root, false);
            if (s == null) {
                zk.create(root, new byte[0], Ids.OPEN_ACL_UNSAFE,
                    CreateMode.PERSISTENT);
            }
        } catch (KeeperException e) {
            System.out
                .println("Keeper exception when instantiating queue: "
                    + e.toString());
        } catch (InterruptedException e) {
            System.out.println("Interrupted exception");
        }
    }
}
```

```

// 节点名称
try {
    name = new String(InetAddress.getLocalHost().getCanonicalHostName().
toString());
} catch (UnknownHostException e) {
    System.out.println(e.toString());
}
}

```

一个进程调用 `enter()` 来进入 `Barrier`。另一个进程在 `root` 下面创建一个节点来代表它，用它的主机名表示节点的名字。然后等待直到足够的进程进入 `Barrier`。一个进程通过 `getChildren()` 检查 `root` 节点的子节点数量来实现，并且等到一旦它没有足够的子节点就能得到通知。为了收到当 `root` 节点改变的时候的通知，进程需要设置一个 `watch`，并且通过调用“`getChildren()`”来实现。在代码中，“`getChildren()`”方法有两个参数。第一个是一个节点需要读取的状态数据，第二个是布尔标识，标明是否设置 `watch`，在代码中该标识是 `true`：

```

/**
 * 加入 Barrier
 *
 * @return
 * @throws KeeperException
 * @throws InterruptedException
 */
boolean enter() throws KeeperException, InterruptedException{
    zk.create(root + "/" + name, new byte[0], Ids.OPEN_ACL_UNSAFE,
        CreateMode.EPHEMERAL_SEQUENTIAL);
    while (true) {
        synchronized (mutex) {
            List<String> list = zk.getChildren(root, true);

            if (list.size() < size) {
                mutex.wait();
            } else {
                return true;
            }
        }
    }
}

```

注意，`enter()`会抛出 `KeeperException` 和 `InterruptedException`，因此应用程序需要自行捕获和处理这种异常。

一旦计算完成，进程调用 `leave()`来离开 `Barrier`。首先，它删除其相应的节点，然后获得 `root` 节点的子节点。如果有至少一个子节点，那么它等待通知。在接收到通知时，它再次检查根节点是否含有任何子节点：

```
/**
 * 等待所有到达的 Barrier
 *
 * @return
 * @throws KeeperException
 * @throws InterruptedException
 */
boolean leave() throws KeeperException, InterruptedException{
    zk.delete(root + "/" + name, 0);
    while (true) {
        synchronized (mutex) {
            List<String> list = zk.getChildren(root, true);
            if (list.size() > 0) {
                mutex.wait();
            } else {
                return true;
            }
        }
    }
}
```

2. Producer-Consumer Queue

生产者-消费者队列是一个数据分发结构，一组进程生产并且消费物品。生产者进程创建新的元素并且加入队列。消费者进程从队列中删除元素，并且处理它们。在这个例子中，元素就是简单的数字。队列以 `root` 节点来表示，并且加入一个元素到这个队列中，一个生产者进程创建一个新节点（`root` 节点的子节点）。

下面的代码含义是：它首先由父类 `SyncPrimitive` 的构造函数来创建一个 `ZooKeeper` 对象，然后校验队列的 `root` 节点是否存在，如果不存在就创建一个：

```
/**
 * producer-consumer queue 构造函数
```

```

*
* @param address
* @param name
*/
Queue(String address, String name) {
    super(address);
    this.root = name;
    // 创建 ZooKeeper 节点名称
    if (zk != null) {
        try {
            Stat s = zk.exists(root, false);
            if (s == null) {
                zk.create(root, new byte[0], Ids.OPEN_ACL_UNSAFE,
                    CreateMode.PERSISTENT);
            }
        } catch (KeeperException e) {
            System.out
                .println("Keeper exception when instantiating queue: "
                    + e.toString());
        } catch (InterruptedException e) {
            System.out.println("Interrupted exception");
        }
    }
}

```

生产者进程调用“produce()”向队列中添加一个元素，并传递一个整数作为参数。向队列添加一个元素时，该方法使用“create()”创建一个新节点，并使用 SEQUENCE 标志指示 ZooKeeper 来附加与根节点关联的序列计数器的值。以这种方式对队列的元素施加总次序，从而保证队列的最旧元素是下一个要消费的元素：

```

/**
 * 添加元素到 queue
 *
 * @param i
 * @return
 */
boolean produce(int i) throws KeeperException, InterruptedException{
    ByteBuffer b = ByteBuffer.allocate(4);
    byte[] value;

```



```

// 添加子节点
b.putInt(i);
value = b.array();
zk.create(root + "/element", value, Ids.OPEN_ACL_UNSAFE,
    CreateMode.PERSISTENT_SEQUENTIAL);

return true;
}

```

为了消费一个元素，消费者进程获得 root 节点的子节点，读取具有最小计数器值的节点，并返回该元素。注意，如果存在冲突，则两个竞争进程之一将不能删除该节点，且删除操作会抛出异常。

getChildren()的调用按词典顺序返回子节点列表。由于词典顺序不必遵循计数器值的数字顺序，因此需要确定哪个元素是最小的。为了确定哪一个具有最小的计数器值，会遍历列表，并删除前缀是“element”的每个元素：

```

/**
 * 移除 queue 中的第一个元素
 *
 * @return
 * @throws KeeperException
 * @throws InterruptedException
 */
int consume() throws KeeperException, InterruptedException{
    int retvalue = -1;
    Stat stat = null;

    // 获取存在的第一个元素
    while (true) {
        synchronized (mutex) {
            List<String> list = zk.getChildren(root, true);
            if (list.size() == 0) {
                System.out.println("Going to wait");
                mutex.wait();
            } else {
                Integer min = new Integer(list.get(0).substring(7));
                for(String s : list){
                    Integer tempValue = new Integer(s.substring(7));

```

```

        //System.out.println("Temporary value: " + tempValue);
        if(tempValue < min) min = tempValue;
    }
    System.out.println("Temporary value: " + root + "/element" + min);
    byte[] b = zk.getData(root + "/element" + min,
        false, stat);
    zk.delete(root + "/element" + min, 0);
    ByteBuffer buffer = ByteBuffer.wrap(b);
    retvalue = buffer.getInt();

    return retvalue;
}
}
}
}

```

上面例子的代码可以在 <https://github.com/waylau/distributed-systems-technologies-and-cases-analysis> 的 zookeeper-demo 程序中找到。

6.6 实战：基于 ZooKeeper 的服务注册和发现

在分布式架构的系统中，系统经常被暴露为服务以供其他系统调用，这也是 SOA 或微服务架构常用的模式。

为了使服务之间互相通信，需要有一个协调系统来管理这些服务，以便这些服务能够互相找到对方，这就是服务注册与发现机制。这个协调系统有时也被称为“注册中心”。

下面，我们将介绍如何基于 ZooKeeper 来实现服务的注册和发现功能。

6.6.1 项目概述

我们将创建一个名为“zk-registry-discovery”的应用，该应用基于 Zookeeper 实现服务的注册和发现功能。

为了能够正常运行该应用，需要在应用中添加如下依赖：

```

<dependencies>
    <dependency>
        <groupId>com.101tec</groupId>
        <artifactId>zkclient</artifactId>
    </dependency>
</dependencies>

```

```

        <version>0.10</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.0.6.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>com.google.guava</groupId>
        <artifactId>guava</artifactId>
        <version>25.0-jre</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>

```

其中，我们采用 ZooKeeper 的 Java 客户端 zkclient，同时使用 Spring、Guava 作为应用的常用工具包。

6.6.2 项目配置

配置 ZooKeeper 的配置文件 zoo.cfg:

```

tickTime=2000
initLimit=10
syncLimit=5
dataDir=D:\\zookeeper\\data
dataLogDir=D:\\zookeeper\\log
clientPort=2181

```

让 ZooKeeper 服务器在 2181 端口启动。为了方便测试，这里仅仅启动了一个 ZooKeeper 节点。

6.6.3 编码实现

定义如下服务注册的接口：

```
public interface ServiceRegistry {  
  
    /**  
     * 注册服务。  
     *  
     * @param serviceName  
     * @param serviceAddress  
     */  
    void registry(String serviceName, String serviceAddress);  
}
```

服务注册的实现如下：

```
package com.waylau.zk.registry;  
  
import org.I0Itec.zkclient.ZkClient;  
  
import com.waylau.zk.Constant;  
  
public class ZkServiceRegistry implements ServiceRegistry {  
  
    /**  
     * ZK 地址  
     */  
    private String zkAddress = "localhost";  
  
    /**  
     * ZK 客户端  
     */  
    private ZkClient zkClient;  
  
    public void init() {  
        zkClient = new ZkClient(zkAddress,  
            Constant.ZK_SESSION_TIMEOUT,  
            Constant.ZK_CONNECTION_TIMEOUT);  
    }  
}
```




```
        System.out.println(">>>connect to zookeeper");
    }

    @Override
    public void registry(String serviceName, String serviceAddress) {
        // 创建 registry 节点 (持久)
        String registryPath = Constant.ZK_REGISTRY;
        if (!zkClient.exists(registryPath)) {
            zkClient.createPersistent(registryPath);

            System.out.println(">>>create registry node:" + registryPath);
        }

        // 创建 service 节点 (持久)
        String servicePath = registryPath + "/" + serviceName;
        if (!zkClient.exists(servicePath)) {
            zkClient.createPersistent(servicePath);
            System.out.println(">>>create service node:" + servicePath);
        }

        // 创建 address 节点 (临时)
        String addressPath = servicePath + "/address-";
        String addressNode = zkClient.createEphemeralSequential(addressPath,
            serviceAddress);

        System.out.println(">>>create address node:" + addressNode);
    }
}
```

同时，我们定义如下服务发现的接口：

```
public interface ServiceDiscovery {

    /**
     * 服务发现
     *
     * @param name
```



```
    * @return
    */
    String discover(String name);
}
```

服务发现的实现如下：

```
package com.waylau.zk.discovery;

import java.util.List;
import java.util.concurrent.ThreadLocalRandom;

import org.I0Itec.zkclient.ZkClient;
import org.springframework.util.CollectionUtils;

import com.google.common.collect.Lists;
import com.waylau.zk.Constant;

public class ZkServiceDiscovery implements ServiceDiscovery {

    /**
     * ZK 地址
     */
    private String zkAddress = "localhost";

    /**
     * 缓存所有的服务 IP 和端口
     */
    private final List<String> addressCache = Lists.newCopyOnWriteArrayList();

    /**
     * ZK 客户端
     */
    private ZkClient zkClient;

    public void init() {
        zkClient = new ZkClient(zkAddress,
            Constant.ZK_SESSION_TIMEOUT,
            Constant.ZK_CONNECTION_TIMEOUT);
    }
}
```



```
        System.out.println(">>>connect to zookeeper");
    }

    @Override
    public String discover(String name) {
        try {
            String servicePath = Constant.ZK_REGISTRY + "/" + name;

            // 获取服务节点
            if (!zkClient.exists(servicePath)) {
                throw new RuntimeException(
                    String.format(">>>can't find any service node on path {}",
                        servicePath));
            }

            // 从本地缓存获取某个服务地址
            String address;
            int addressCacheSize = addressCache.size();
            if (addressCacheSize > 0) {
                if (addressCacheSize == 1) {
                    address = addressCache.get(0);
                } else {
                    address = addressCache.get(ThreadLocalRandom.current().
nextInt(addressCacheSize));

                    System.out.println(">>>get only address node:" + address);
                }

                // 从 zk 服务注册中心获取某个服务地址
            } else {
                List<String> addressList = zkClient.getChildren(servicePath);
                addressCache.addAll(addressList);

                // 监听 servicePath 下的子文件是否发生变化
                zkClient.subscribeChildChanges(servicePath, (parentPath,
currentChilds) -> {
                    System.out.println(">>>servicePath is changed:" + parentPath);

                    addressCache.clear();
                });
            }
        }
    }
}
```



```

        addressCache.addAll(currentChilds);
    });

    if (CollectionUtils.isEmpty(addressList)) {
        throw new RuntimeException(
            String.format(">>>can't find any address node on
path {}", servicePath));
    }

    int nodes = addressList.size();
    if (nodes == 1) {
        address = addressList.get(0);
    } else {

        // 如果多个，则随机取一个
        address = addressList.get(ThreadLocalRandom.current().
nextInt(nodes));
    }

    System.out.println(">>>get address node:" + address);
}

// 获取 IP 和端口号
String addressPath = servicePath + "/" + address;
String hostAndPort = zkClient.readData(addressPath);
return hostAndPort;
} catch (Exception e) {

    System.out.println(">>>service discovery exception" + e.getMessage());

    zkClient.close();
}
return null;
}
}

```

以下是两个服务公用的常量：

```
public interface Constant {
```




```
/**会话超时时间*/  
int ZK_SESSION_TIMEOUT = 5000;  
  
/**连接超时时间*/  
int ZK_CONNECTION_TIMEOUT = 1000;  
  
String ZK_REGISTRY = "/registry";  
}
```

6.6.4 运行

为了方便测试，我们编写了如下测试用例：

```
package com.waylau.zk;  
  
import org.junit.Test;  
  
import com.waylau.zk.discovery.ZkServiceDiscovery;  
import com.waylau.zk.registry.ZkServiceRegistry;  
  
public class ApplicationTests {  
  
    private static final String SERVER_NAME = "waylau.com";  
    private static final String SERVER_ADDRESS = "localhost:2181";  
  
    @Test  
    public void testClient() throws Exception {  
  
        ZkServiceRegistry registry = new ZkServiceRegistry();  
        registry.init();  
        registry.registry(SERVER_NAME, SERVER_ADDRESS);  
  
        ZkServiceDiscovery discovery = new ZkServiceDiscovery();  
        discovery.init();  
        discovery.discover(SERVER_NAME);  
  
        // 永不停止  
        while(true) {
```



```
}  
  
}  
  
}
```

先启动 ZooKeeper 服务，再执行该测试用例。我们分别启动三个测试用例，以模拟多个客户端同时进行注册服务的场景。程序执行后，观察控制台的输出信息。

第一个测试用例的输出如下：

```
>>>connect to zookeeper  
>>>create registry node:/registry  
>>>create service node:/registry/waylau.com  
>>>create address node:/registry/waylau.com/address-0000000000  
>>>connect to zookeeper  
>>>get address node:address-0000000000  
>>>servicePath is changed:/registry/waylau.com  
>>>servicePath is changed:/registry/waylau.com
```

第二个测试用例的输出如下：

```
>>>connect to zookeeper  
>>>create address node:/registry/waylau.com/address-0000000001  
>>>connect to zookeeper  
>>>get address node:address-0000000001  
>>>servicePath is changed:/registry/waylau.com
```

第三个测试用例的输出如下：

```
>>>connect to zookeeper  
>>>create address node:/registry/waylau.com/address-0000000002  
>>>connect to zookeeper  
>>>get address node:address-0000000001
```

从上面例子运行的结果可以看出，第一个测试用例先运行，而其他服务还没有注册，所以在获取可用的服务时获取了自己。第二个和第三个测试用例运行后，可用的服务实例就有多个，所以在获取服务时，有可能获取自己，也可能获取其他服务实例。当有新的服务进行注册时，所有的服务实例都能感知到新服务的加入。

上述代码可以在 <https://github.com/waylau/distributed-systems-technologies-and-cases-analysis> 的 zk-registry-discovery 程序中找到。





7 chapter

第 7 章

分布式版本控制系统





7.1 分布式版本控制系统概述

在企业中，项目源码或文档往往需要进行版本的管理。即便是个人的工作，采用版本管理工具进行管理，对于方便查找特定版本的内容，或者回溯历史的修改内容都是极其必要的。

版本控制系统是帮助人们协调工作的工具，它能够帮助我们和其他小组成员监测同一组文件，比如软件源代码、升级过程中所做的变更等。也就是说，版本控制系统可以帮助我们轻松地将工作进行融合。

7.1.1 集中式与分布式

许多传统的版本控制工具都需要一个中心服务器来存放一系列文件及其更新日志。为了工作，用户必须连接服务器并校验文件，因此必须有一个用户能更改的目录或工作树。为了能够记录或提交他们的工作，用户必须连接到中心服务器并且确保他们的工作能够与试图提交前的最新版本兼容，这就是所谓的集中式。

时间证明了集中式是有效的，但也显露了明显的缺点。首先，集中式的版本控制系统必须确保当某个用户想要对版本进行修改时可以随时连接到服务器。其次，集中式必须紧紧地依赖修改和发布的行为，这在某些情况下是好的，但也有可能影响其他人的工作质量。

分布式版本控制系统允许用户和团队拥有多个目录而不是只有一个中心目录。在版本控制系统中，日志通常保存在同一个位置，以此来对代码进行版本控制。这样用户就可以在代码有效的情况下随时提交他们的代码，即使是离线状态。只有当发布修改和读取其他位置的修改时，网络连接才是必需的。

事实上，若开发者对分布式版本控制系统使用得当，则可以获得远远超过之前离线操作的优势。这些优势包括：

- 更容易地创建实验分支；
- 极其容易地与他人合作；
- 节约机械式任务的时间；
- 可以有更多的时间来创造；
- 通过使用“广泛特征”协议来不断提高发布管理的灵活性；
- 主版本的质量和稳定性会更好，减小了所有人的压力；
- 在开放的社区中
 - 方便非核心开发者创建和维护更新；





- 方便核心开发者与非核心开发者之间的合作并吸纳他们；
- 在公司中，分散的和外包的队伍的工作将会更方便。

7.1.2 分布式版本控制系统的核心概念

分布式版本控制系统主要包含以下四个核心概念：

- Revision（修订版）——所修改的文件的快照（snapshot）；
- Working tree（工作树）——存储需要做版本控制的文件的目录和子目录；
- Branch（分支）——按顺序存放的修订版来记录一系列文件的修改历史；
- Repository（仓库）——存放修订版的存储库。

目前，市面上流行的分布式版本控制系统主要有 Bazaar、Mercurial 和 Git 等，但不管是哪种技术，都包含上述四个核心概念。

7.2 Bazaar

Bazaar 是一个分布式的版本控制系统，采用 GPL 许可协议，由 Canonical 公司（Ubuntu 母公司）赞助。Bazaar 是 Python 编写的，可运行于 Windows、GNU/Linux、UNIX 和 Mac OS 系统之上。

7.2.1 Bazaar 的核心概念

Bazaar 主要包含以下四个核心概念。

1. Revision

一个修订版就是一系列文件和目录的独立镜像，包括它们的内容和大小。一个修订版还有一些与之相关的属性，包括：

- 提交者；
- 提交时间；
- 提交的信息；
- 来源于哪一个修订版。

修订版是不可改变的，并且可以有全局唯一的修订 id 号。一个修订 id 号类似于如下格式：

```
pqm@pqm.ubuntu.com-20071129184101-u9506rihe4zbzyyz
```





修订 id 号是在提交或从其他系统中转移时生成的。尽管修订 id 号对内部使用和与外部工具的融合是必需的，但特定分支的修订号对人来说是界面友好的。

修订号是用点来分隔的十进制数，例如 1.42 和 2977.1.59，这一系列数都记录了某一个分支的发展过程。一般情况下，修订号都会比修订 id 号短，并且在某一个独立的分支内部可以通过比较得出两个修订版之间的关系。比如说，修订版 10 是修订版 9 之后短时间内的主分支版本。修订号在命令执行之前就已经生成，因为修订号依赖于分支顶端（也就是最新的）修订版。

2. Working Tree

一个工作树就是一个存放用户能够编辑的文件的版本收集目录，并且会与特定的一个分支结合在一起。

很多命令会把当前工作树作为默认环境使用，比如 `commit` 命令会在当前工作树中创建一个新的修订版。

3. Branch

简单地说，一个分支就是一系列有序的修订版。最新的修订版被称作顶端（tip）。

多个分支既可以被分割开，也可以重新合并（merged）共同组成一个修订版的图（graph）。专业点说，这个图应当是有向的（用以展示双亲修订版和儿子修订版的关系）并且是无环的，也就是常说的有向无环图（directed acyclic graph，简称 DAG）。

如果这个名字听起来有点吓人，没关系，只需要记住：

- DAG 中最早的发展路径被称作主分支（mainline）、主干（trunk）或左手边（left hand side，简称 LHS）；
- 一个分支也许会有另一条发展路径，如果有，那么这条路径一定是在某个节点开始，在某一个节点结束。

4. Repository

仓库就是存储修订版文件的地方。在最简单的情况下，每个分支都有它自己的仓库。而在其他情况下，也会为了有效地使用硬盘而出现多个分支共享同一个仓库的情况。

7.2.2 Bazaar 的使用

本节主要介绍 Bazaar 的基本使用方法。

1. 配置用户名

用户名（一般就是 E-mail 地址）是 Bazaar 的唯一身份识别，它有以下几种配置方式。



(1) 通过 `bzr whoami` 来设置。

这是最简单的设置全局唯一账号的方式，使用：

```
% bzr whoami "Your Name <email@example.com>"
```

若想给不同的分支设置不同的账号，则使用：

```
% bzr whoami --branch "Your Name <email@example.com>"
```

(2) 在 `~/bazaar/bazaar.conf` 中设置。

在 `~/bazaar/bazaar.conf` 中的设置如下：

```
[DEFAULT]
email=Your Name <email@isp.com>
```

若想给不同的分支设置不同的账号，则需在 `~/bazaar/bazaar.conf` 中增加配置项：

```
[/the/path/to/the/branch]
email=Your Name email@isp.com
```

(3) 设置全局环境变量。

将全局环境变量 `$BZR_EMAIL` 或 `$EMAIL` (`$BZR_EMAIL` 将优先) 设置为 E-mail。

2. 创建分支

历史记录默认存储在分支的 `.bzr` 目录中。在 Bazaar 的未来版本中，有一个工具将它存储在一个单独的存储库中，这可能是远程的。

通过在现有目录中运行 `bzr init` 来创建一个新的分支：

```
% mkdir tutorial
% cd tutorial
% ls -a
./ ../
% pwd
/home/mbp/work/bzr.test/tutorial
%
% bzr init
% ls -aF
./ ../ .bzr/
%
```

与一般的 CVS 一样，Bazaar 主要有三类文件：未知、被忽略和版本化。`add` 命令使文件版本化，即对该文件的所有更改都会记录到系统中：



```
% echo 'hello world' > hello.txt
% bzip status
unknown:
  hello.txt
% bzip add hello.txt
added hello.txt
% bzip status
added:
  hello.txt
```

如果版本化了错误的文件，可以使用 `bzip remove` 来取消版本化。

3. 分支位置

所有历史记录都被存储在一个分支中，它只是一个包含控制文件的磁盘目录。在默认情况下不存在 `svn` 或 `svk` 中使用的单独的存储库或数据库。如果有需要，则可以选择创建存储库（请参阅 `bzip init-repo` 命令），这通常适用于非常大的分支，或者拥有许多分支的中等规模的项目。

通常，通过提供包含分支的目录的名称来引用计算机文件系统上的分支。`bzip` 还支持通过 `SSH`、`HTTP` 和 `SFTP` 的访问分支：

```
% bzip log bzip+ssh://bazaar.launchpad.net/~bzip-pqm/bzip/bzip.dev/
% bzip log http://bazaar.launchpad.net/~bzip-pqm/bzip/bzip.dev/
% bzip log sftp://bazaar.launchpad.net/~bzip-pqm/bzip/bzip.dev/
```

通过安装 `bzip` 插件，还可以使用 `rsync` 协议来访问分支。

4. 查看更改

一旦完成了一些工作，比如完成了一个新功能，修复一个错误，或者改进一些代码或文档，就都需要 `commit`（提交）它的版本历史。频繁提交更改是一个好的做法，这样可以确保每个修订版本都是已知的良好状态。还可以查看更改信息，以确保该更改符合我们的意图，然后永久记录。

有两个 `bzip` 命令在这里特别有用：`status` 和 `diff`。

（1）`bzip status`。

`status` 命令告诉我们自上次修订以来对工作目录所做的更改：

```
% bzip status
modified:
  foo
```

（2）`bzip diff`。

`diff` 命令显示所有文件的更改：




```
% bzip diff
=== added file 'hello.txt'
--- hello.txt    1970-01-01 00:00:00 +0000
+++ hello.txt    2005-10-18 14:23:29 +0000
@@ -0,0 +1,1 @@
+hello world
```

使用 `-r` 选项，将树与较早的版本进行比较，或者显示两个版本之间的差异：

```
% bzip diff -r 1000..          # 从 1000 版本开始
% bzip diff -r 1000..1100     # 从 1000 到 1100 之间的版本
```

使用 `--diff-options` 选项使 `bzip` 运行外部 `diff` 程序、传递选项。例如：

```
% bzip diff --diff-options --side-by-side foo
```

5. 提交更改

使用 `commit` 命令来提交变更，增加 `-m` 或 `--message` 选项可以在提交时增加备注信息：

```
% bzip commit -m "added my first file"
```

标记 bug 已修复

使用 `--fixes` 选项来标记一个提交或者一个 bug 修复，比如：

```
% bzip commit --fixes <tracker>:<id>
```

其中，`<tracker>` 是 bug tracker 的唯一标识，`<id>` 是 bug tracker 中 bug 的唯一标识。

下面是一个完整示例：

```
% bzip commit -m "fixed my first bug" --fixes lp:1234
```

选择性提交

可以选定特定文件或者目录来提交，例如：

```
% bzip commit -m "documentation fix" commit.py
```

提交当前目录，执行：

```
% bzip commit .
```

移除未提交的变更

如果所做的变更不想保持，则可以通过执行 `bzip revert` 来还原。

6. 过滤文件

可以设置过滤策略，这样就能在版本化时排除这些文件。



.bzrignore 文件

配置策略可以配置在.bzrignore 文件中。以下是全局的过滤：

```
*.o
*~
*.tmp
*.py[co]
```

以下是匹配特定路径的过滤：

```
./config.h
doc/*.html
```

列出被过滤的文件：

```
% bzip ignored
config.h                ./config.h
configure.in~          *~
```

bzip ignore

执行 bzip ignore，通过命令行在.bzrignore 文件中添加过滤的策略：

```
% bzip ignore tags
% bzip status
added:
    .bzrignore
```

全局过滤

全局过滤适用于全部项目，过滤文件在 ~/.bazaar/ignore 中。

7. 检查历史

执行 bzip log 命令用于显示以前版本的列表。bzip log -forward 命令按照时间顺序执行，以获取最后打印的最近修订版本。

与 bzip diff 一样，bzip log 也支持 -r 参数：

```
% bzip log -r 1000..      # 从 1000 版本开始
% bzip log -r ..1000      # 1000 版本之前的（包括 1000 版本）所有版本
% bzip log -r 1000..1100  # 从 1000 到 1100 之间的版本
% bzip log -r 1000        # 只是 1000 版本
```

8. 分支统计

bzip info 命令可以显示有关工作树和分支历史记录的一些摘要信息。



9. 目录的版本控制

对文件和目录进行版本化，可以跟踪文件或目录的重命名，并能智能地合并它们：

```
% mkdir src
% echo 'int main() {}' > src/simple.c
% bzip add src
added src
added src/simple.c
% bzip status
added:
  src/
  src/simple.c
```

10. 分支化

复制当前的分支来生成一个新的分支：

```
% bzip branch lp:bzip bzip.dev
% cd bzip.dev
```

11. 跟随上游更改

可以通过“pull”获取父分支来保持最新的更新：

```
% bzip pull
```

12. 合并相关的分支

合并分支，执行如下：

```
% bzip merge URL
```

13. 发布分支

发布分支，执行如下：

```
% bzip push bzip+ssh://servername.com/path/to/directory
```

7.3 Mercurial

Mercurial 是一个免费的分布式源代码控制管理工具。它可以有效地处理任何大小的项目，并提供一个简单和直观的界面。Mercurial 支持 Windows 及类 UNIX 系统，比如 FreeBSD、Mac OS X 和 Linux。



7.3.1 Mercurial 的核心概念

本节介绍 Mercurial 的核心概念。

1. Repository

Mercurial repository（仓库）包含一个工作目录和一个存储，如图 7-1 所示。

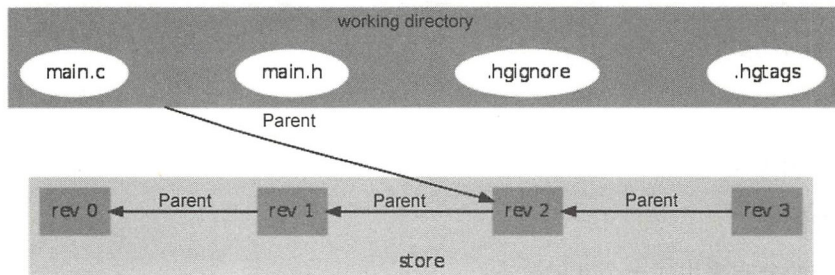


图 7-1 Mercurial repository

存储包含项目的完整历史记录。与传统的 SCM 不同，传统的 SCM 只有一个历史记录的中心副本，每个工作目录都与历史的私人副本配对。这使得开发可以并行进行。

工作目录包含项目文件在给定时间点的副本（如 rev 2），以备编辑。因为标签文件（.hgtags）和过滤策文件（.hignore）也是受版本控制的，所以也包括在内。

2. 提交更改

当提交时，工作目录相对于其父节点的状态被记录为新的变更集（changeset），也称为新的“修订版（revision）”，如图 7-2 所示。

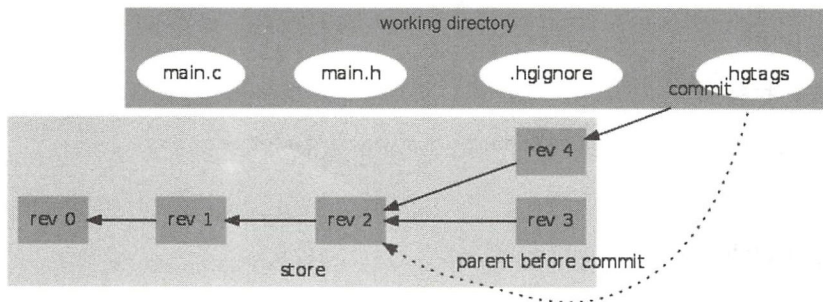


图 7-2 Mercurial 提交更改

注意，修订版本 rev 4 是修订版本 rev 2 的一个分支，它是工作目录中的修订版本。现在 rev 4 是工作目录的父目录。

3. revision、changeset、head 和 tip

Mercuria 将与多个文件相关的更改归类为单个原子变更集（changeset），这是整个项目的修订版本。这些都会获得一个连续的修订号。因为 Mercurial 允许分布式并行开发，这些修订版本号可能在用户之间不一致。因此，Mercurial 还为每个修订版本分配了一个全局 changeset ID。changeset ID 是 40 位十六进制数字，但它们可以缩写为任何明确的前缀，例如“e38487”。Mercurial repository 示意图如图 7-3 所示。

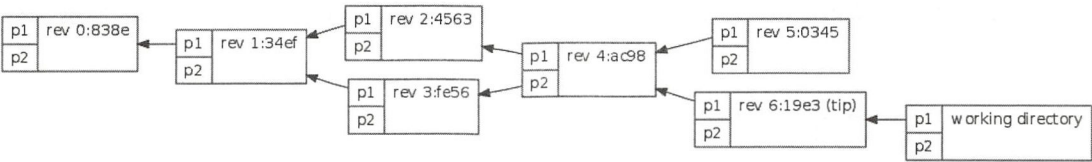


图 7-3 Mercurial repository 示意图

Head 表示 repository 中每个分支最新的 revision，通常在合并几个分支时会用到这个概念。

tip 是最新的一个 changeset 的版本号的一个别名。在命令中任何使用版本号的地方都可以使用 tip 来代替最新的 changeset 的版本号。tip 在各个 repository 中是不同的，同时一个 repository 中只有一个 tip。

4. clone、标记更改、merge、pull 和 update

假设有一个用户 Alice，她有一个存储仓库，如图 7-4 所示。

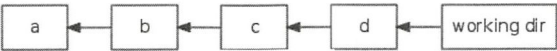


图 7-4 Alice 的 repository

Bob 克隆（clone）这个仓库，最终得到一个完整的、独立的、本地的 Alice 存储的副本，并将一个干净的顶端的修订 d 检出到她的工作目录，如图 7-5 所示。

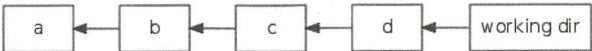


图 7-5 Bob 的 repository

Bob 现在可以独立于 Alice 工作。然后他提交两个变化 e 和 f，如图 7-6 所示。

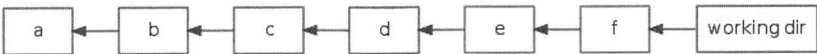


图 7-6 Bob 的 repository

然后，Alice 并行地进行自己的更改，这导致她的存储库与 Bob 分开了，从而创建一个分支，如图 7-7 所示。

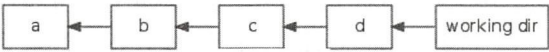


图 7-7 Alice 的 repository

Bob “pull”了 Alice 的仓库来同步。所有 Alice 的更改将复制到 Bob 的存储库进行存储（这里，它只是单一地更改 g）。注意，Bob 的工作目录不会因为 pull 而改变，如图 7-8 所示。

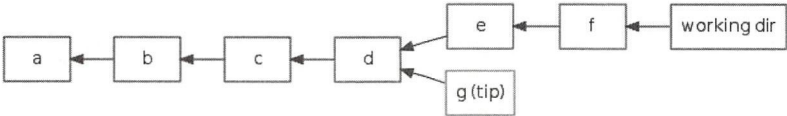


图 7-8 Bob 的 repository

因为 Alice 的 g 是 Bob 的存储库中的最新 head，所以现在是 tip。

然后 Bob 进行合并，将他正在进行的最后一次更改 f 与其存储库中的 tip 相结合。现在，他的工作目录有两个父版本 f 和 g，如图 7-9 所示。

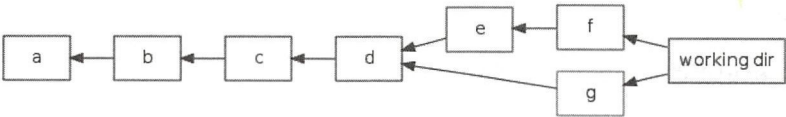


图 7-9 Bob 的 repository

检查工作目录中合并的结果并确保合并是完美的之后，Bob 提交结果，并在存储中产生新的合并变更集 h，如图 7-10 所示。

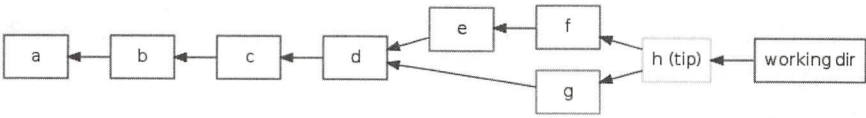


图 7-10 Bob 的 repository

现在如果 Alice 从 Bob “pull”项目， she 会把 Bob 的变化 e、f 和 h 更新到自己的存储中，如图 7-11 所示。

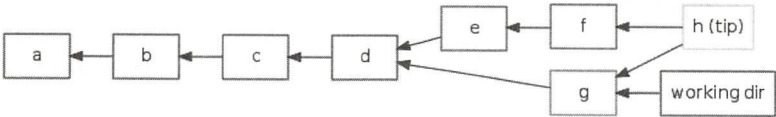


图 7-11 Alice 的 repository

注意, Alice 的工作目录没有因为 pull 而改变, 她必须进行更新以将她的工作目录同步这个合并变更集 h, 这会将她的工作目录的父更改集更改为更改集 h, 并将她工作目录中的文件更新为修订版 h, 如图 7-12 所示。

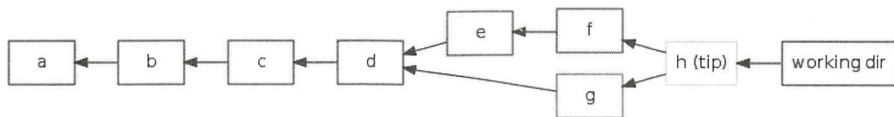


图 7-12 Alice 的 repository

现在, Alice 和 Bob 再次完全同步。

5. 一个分散的系统

Mercurial 是一个完全分散的系统, 因此没有一个中央存储库的概念。用户可以自由定义自己的拓扑以共享变更, 如图 7-13 所示。

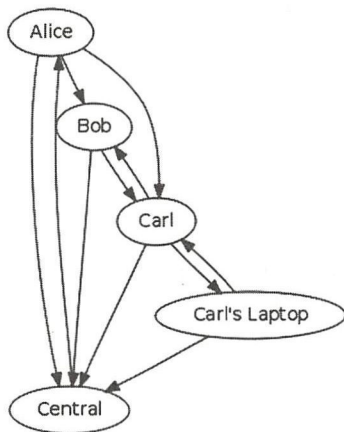


图 7-13 Mercurial 工作网络示意图

7.3.2 Mercurial 的使用

本节介绍 Mercurial 不同的工作流程。通过工作流程来学习 Mercurial 的使用。

1. 基本工作流程

日志记录

第一个工作流程是最简单的, 通过 Mercurial 回顾之前的更改历史。

(1) 准备 Mercurial。

编辑 hgrc 文件 (在 Windows 中是 .ini 文件), 添加用户名:

```
[ui]
username = Mr. Johnson <johnson@smith.com>
```

（2）初始化项目。

添加一个项目文件夹：

```
$ hg init project
```

（3）添加文件并追踪修改。

进入项目文件夹，创建文件并提交它们：

```
$ cd project
$ echo 'print("Hello")' > hello.py
$ hg add
$ hg commit
```

这样，就能通过 `hg log` 命令查看初始化的历史记录：

```
$ hg log
```

（4）保存修改。

首先，做一些更改：

```
$ echo 'print("Hello World")' > hello.py
```

查看哪些文件被修改，哪些被添加或删除到版本中，以及还有哪些没有被版本记录：

```
$ hg status
```

```
M hello.py
```

查看确切的改动：

```
$ hg diff
```

```
diff --git a/hello.py b/hello.py
--- a/hello.py
+++ b/hello.py
@@ -1,1 +1,1 @@
-print("Hello")
+print("Hello World")
```

向 Mercurial 提交更改：

```
$ hg commit
```



```
diff --git a/hello.py b/hello.py
--- a/hello.py
+++ b/hello.py
@@ -1,1 +1,1 @@
-print("Hello")
+print("Hello World")
```

(5) 移动和复制文件。

当移动或复制文件时，一定要使用 Mercurial，从而能够记录这些文件之间的关系。

记住在移动和复制之后要 commit。在基本命令中，只有 commit 可以产生新的版本。

```
$ hg cp hello.py copy
$ hg mv hello.py target
$ hg diff # 查看变更
$ hg commit
```

现在有两个文件，“copy”和“target”，Mercurial 知道它们之间是如何联系起来的。

```
diff --git a/hello.py b/copy
rename from hello.py
rename to copy
diff --git a/hello.py b/target
copy from hello.py
copy to target
```

(6) 查看历史记录。

执行：

```
$ hg log
```

上述操作将打印出所有的版本变化和它们的提交日期、提交者和版本信息。

```
changeset: 2:70eb0ca9d264
tag:      tip
user:     Mr. Johnson
date:     Sun Nov 20 11:20:00 2011 +0100
summary:  Copy and move.

changeset: 1:487d7a20ccbc
user:     Mr. Johnson
```

```
date:      Sun Nov 20 11:11:00 2011 +0100
summary:   Say Hello World, not just Hello.

changeset: 0:a5ecbf5799c8
user:      Mr. Johnson
date:      Sun Nov 20 11:00:00 2011 +0100
summary:   Initial commit.
```

可以使用 `-r` 选项（同 `--revision`）来查看特定的版本。查看显示的版本间的差异可以使用 `-p` 选项（同 `--patch`）：

```
$ hg log -p -r 3
```

单个开发者和非线性开发历史

第二个工作流程仍然很简单：一个独立的开发者打算用 Mercurial 记录自己的版本变化。这和日志记录流程有些类似，不同之处在于有时会回到一个早期版本。

同样建立新项目，初始化 repository，添加文件并提交。也要不时地检查历史记录，看看是怎么一步一步开发到现在的。

整个工作流程如下。

（1）和日志记录一样的基础命令。

初始化项目，添加、查看更改、提交文件：

```
$ hg init project
$ cd project
$ (add files)
$ hg add # tell Mercurial to track all files
$ (do some changes)
$ hg diff # see changes
$ hg commit # save changes
$ hg cp # copy files or folders
$ hg mv # move files or folders
$ hg log # see history
```

（2）查看一个早期版本。

和日志记录不同，我们希望回到某一个早期版本并且直接在那个版本中修改。例如，因为一个早期的修改引入了一个 bug，我们希望修正它。

可以使用 `update` 来查看早期版本的代码。假设要看版本 1，执行：

```
$ hg update 1
```

现在代码就返回了版本 1 的状态，即第 2 次提交（从 0 开始计算）。用 `identify -n` 确认所在的版本：

```
$ hg identify -n
```

```
1
```

注意：当在最新的版本时，`hg identify -n` 将返回“-1”。

可以使用“tip”作为版本名来“update”到最新的版本：

```
$ hg update tip
```

（3）修正早期版本中的错误。

当发现一个早期版本中的 bug 时，有两种选择：直接在当前版本中修改，也可以回到历史版本中修改，从而让历史记录更加清晰。

为了让历史记录更清晰，首先要“update”到过去的版本，修正 bug 并提交。之后将整个版本“merge”到新版本中并提交。别担心，在 Mercurial 中 merge 操作是快速而轻松的。

首先假设 bug 在版本 1 中被引入。

```
$ hg update 1
$ echo 'print("Hello Mercurial")' > hello.py
$ hg commit
```

现在修正已经保存在历史版本中。我们只需要将它和最新版本“merge”：

```
$ hg merge
```

```
merging hello.py and copy to copy
merging hello.py and target to target
```

如果版本间有冲突，则用 `hg resolve` 解决，当然 merge 命令在出现冲突时会有提示。

首先列出冲突的文件：

```
hg resolve -list
```

然后解决它们，用 `resolve` 尝试再次“merge”：

```
$ hg resolve conflicting_file
```

标记修正的文件已经解决了：

```
$ hg resolve --mark conflicting_file
```

一旦解决了所有的冲突，提交 merge。即使没有冲突，这一步也是必要的：

```
$ hg commit
```

在使用时，修正会被应用到所有其他的工作中，然后我们就可以继续编码了。另外，历史记录将清晰地显示在哪里修复了 bug，所以我们可以一直核对是哪里的 bug：

```
changeset: 4:3b06bba7c1a9
tag:       tip
parent:    3:7ff5cd572d80
parent:    2:70eb0ca9d264
user:      Mr. Johnson
date:      Sun Nov 20 20:11:00 2011 +0100
summary:   merge greeting and copy+move.
```

```
changeset: 3:7ff5cd572d80
parent:    1:487d7a20ccbc
user:      Mr. Johnson
date:      Sun Nov 20 20:00:00 2011 +0100
summary:   Greet Mercurial
```

```
changeset: 2:70eb0ca9d264
user:      Mr. Johnson
date:      Sun Nov 20 11:20:00 2011 +0100
summary:   Copy and move.
```

```
changeset: 1:487d7a20ccbc
user:      Mr. Johnson
date:      Sun Nov 20 11:11:00 2011 +0100
summary:   Say Hello World, not just Hello.
```

```
changeset: 0:a5ecbf5799c8
user:      Mr. Johnson
date:      Sun Nov 20 11:00:00 2011 +0100
summary:   Initial commit.
```

2. 高级工作流程

回退错误的修订版本

回退更改意味着创建一个用于将坏的更改逆转的提交。这样虽然无法摆脱历史上的坏代码，

但可以在新的修订中将它删除。

假设坏的更改在修订版 3 中，并且存储库中已经有一个修订版本。要删除错误的代码，可以回退它。这里创建一个新的变更，逆转了坏的变更。回退后，可以将新的更改合并到当前代码。

```
$ hg backout 3
$ hg merge
(potentially resolve conflicts)
$ hg commit
(enter commit message. For example: "merged backout")
```

至此已经逆转了坏的变更，在历史记录里仍然有它的记录，但它不会影响未来的代码。

功能的协作开发

当要把开发分为几个功能时，我们需要跟踪谁在使用哪个功能，哪里得到了变更。

注：clone 的仓库总是先处于默认分支上。

(1) 在命名的分支上工作。

创建分支：

```
$ hg branch feature1
(do some changes)
$ hg commit
(write commit message)
```

更新分支：

```
$ hg update feature1
(do some changes)
$ hg commit
(write commit message)
```

(2) 在命名的分支上做合并。

当完成了一个功能后，可以“merge”该分支到默认的分支：

```
$ hg update default
$ hg merge feature1
$ hg commit
(write commit message)
```

也可以将完成功能的分支关闭，执行：

```
$ hg update feature1
$ hg commit --close-branch -m "finished feature1"
```

7.4 Git

Git 是一款免费、开源的分布式版本控制系统，用于敏捷高效地处理任何或大或小的项目。

在国内，下载 Git 有时很慢。笔者创建了一个开源的项目 `git-for-win`，用于维护最新 Git 客户端的国内下载地址，网址为 <https://github.com/waylau/git-for-win>。

7.4.1 Git 的基础概念

下面介绍 Git 的思想和基本工作原理，这样在使用时 Git 就会知其所以然，游刃有余。

Scott Chacon 在 *Pro Git* 一书中对 Git 的思想和基本工作原理作了如下所示的总结。

1. 直接记录快照，而非差异比较

Git 和其他版本控制系统（包括 Subversion 和近似工具）的主要差别在于 Git 对待数据的方法。从概念上来区分，其他大部分系统以文件变更列表的方式存储信息，这类系统（CVS、Subversion、Perforce、Bazaar 等）将它们保存的信息看作一组基本文件和每个文件随时间逐步累积的差异，如图 7-14 所示。

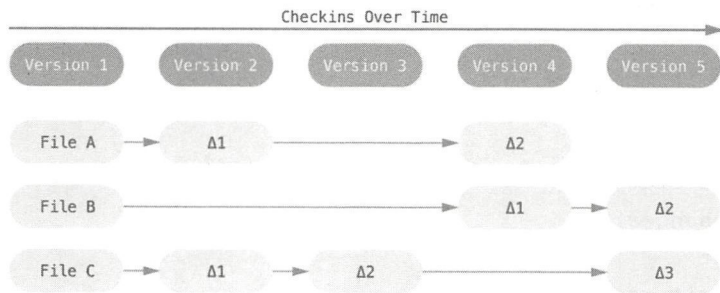


图 7-14 存储每个文件与初始版本的差异

Git 不按照以上方式对待或保存数据。相反，Git 更像是把数据看作对小型文件系统的一组快照。每次提交更新，或在 Git 中保存项目状态时，它主要对当时的全部文件制作一个快照并保存这个快照的索引。为了高效，如果文件没有修改，则 Git 不再重新存储该文件，而是只保留一个链接指向之前存储的文件。Git 对待数据更像是一个快照流（a stream of snapshots），如图 7-15 所示。

这是 Git 与几乎所有其他版本控制系统的重要区别。因此 Git 重新考虑了以前每一代版本控制系统延续下来的诸多方面。Git 更像是一个小型的文件系统，提供了许多以此为基础构建的超强工具，而不只是一个简单的 VCS。

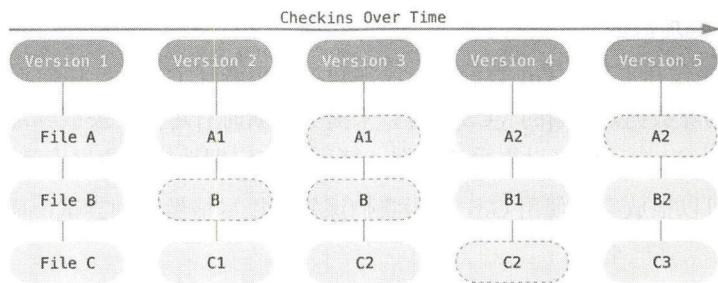


图 7-15 存储项目随时间改变的快照

2. 几乎所有操作都是本地执行的

在 Git 中的绝大多数操作都只需要访问本地文件和资源，一般不需要来自网络上其他计算机的信息。如果习惯于所有操作都有网络延时开销的集中式版本控制系统，则 Git 在这方面会让你感到极其快速。因为在本地磁盘上就存有项目的完整历史，所以大部分操作看起来是瞬间完成的。

举个例子，要浏览项目的历史，Git 不需外连服务器去获取历史，然后显示出来——它只需直接从本地数据库中读取，我们就能立即看到项目历史。如果想查看当前版本与一个月前的版本之间引入的修改，则 Git 会查找到一个月前的文件做一次本地的差异计算，而不是由远程服务器处理或从远程服务器拉回旧版本文件再来本地处理。

这也意味着离线时几乎可以进行任何操作。比如，在飞机或火车上想做些工作，可以先进行提交，直到有网络连接时再上传。而其他系统，做到如此几乎是不可能的。比如用 Perforce，没有连接服务器时几乎不能做什么事；用 Subversion 和 CVS，能修改文件，但不能向数据库提交修改（因为本地数据库离线了）。

3. Git 能保证完整性

Git 中所有的数据在存储前都计算校验和，然后以校验和来引用，这意味着不可能在 Git 不知情时更改任何文件内容或目录内容。这个功能建构在 Git 底层，是构成 Git 哲学不可或缺的部分。若在传送过程中丢失信息或损坏文件，则 Git 就能发现。

Git 用以计算校验和的机制叫作 SHA-1 散列（hash，哈希）。这是一个由 40 个十六进制字符（0~9 和 a~f）组成的字符串，基于 Git 中文件的内容或目录结构计算出来。SHA-1 哈希看起来是这样的：

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Git 中使用这种哈希值的情况很多，我们将经常看到这种哈希值。实际上，Git 数据库中保存的信息都是以文件内容的哈希值来索引的，而不是文件名。

4. Git 一般只添加数据

执行的 Git 操作几乎只往 Git 数据库中添加数据。很难让 Git 执行任何不可逆操作，或者让它以任何方式清除数据。同别的 VCS 一样，未提交更新时有可能丢失或弄乱修改的内容；但是一旦提交快照到 Git 中，就难以再丢失数据，特别是在定期地推送数据库到其他仓库时。

这使得我们可以很放心地使用 Git，因为我们深知可以尽情做各种尝试，而没有把事情弄糟的危险。

5. 三种状态

Git 有三种状态：已提交（committed）、已修改（modified）和已暂存（staged）。已提交表示数据已经安全地保存在本地数据库中。已修改表示修改了文件，但还没保存到数据库中。已暂存表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中。

由此引入 Git 项目的三个工作区域的概念：Git 仓库（directory 或 repository）、工作目录（working directory）与暂存区域（staging area），如图 7-16 所示。

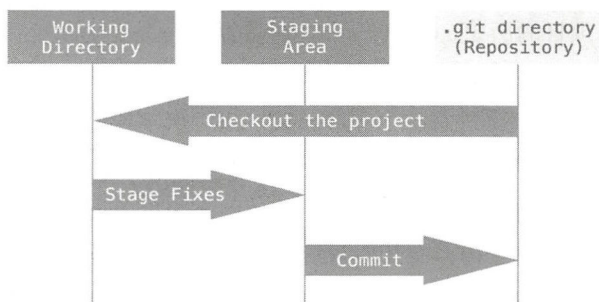


图 7-16 工作目录、暂存区域与 Git 仓库

Git 仓库目录是 Git 用来保存项目的元数据和对象数据库的地方。这是 Git 中最重要的部分，从其他计算机克隆仓库时，复制的就是这里的数据。

工作目录是对项目的某个版本独立提取出来的内容。这些从 Git 仓库的压缩数据库中提取出来的文件放在磁盘上供我们使用或修改。

暂存区域是一个文件，保存下次将提交的文件列表信息，一般在 Git 仓库目录中。有时候也被称作“索引（index）”，不过一般说法还是叫暂存区域。

基本的 Git 工作流程如下：

- 在工作目录中修改文件；
- 暂存文件，将文件的快照放入暂存区域；
- 提交更新，找到暂存区域的文件，将快照永久性存储到 Git 仓库目录。

如果 Git 目录中保存着特定版本文件，则属于已提交状态。如果做了修改并已放入暂存区域，则属于已暂存状态。如果自上次取出后，做了修改但还没有放到暂存区域，则是已修改状态。

7.4.2 Git 的使用

本节内容将介绍在使用 Git 完成各种工作中要使用的各种基本命令。

1. 获取 Git 仓库

有两种取得 Git 项目仓库的方法：第一种是在现有项目或目录下导入所有文件到 Git 中；第二种是从一个服务器复制一个现有的 Git 仓库。

(1) 在现有目录中初始化仓库。

如果打算使用 Git 来对现有的项目进行管理，则只需要进入该项目目录并输入：

```
$ git init
```

该命令将创建一个名为 `.git` 的子目录，这个子目录含有初始化的 Git 仓库中所有的必需文件，这些文件是 Git 仓库的骨干。但是，在这个时候，我们仅仅做了一个初始化的操作，项目里的文件还没有被跟踪。

如果是在一个已经存在文件的文件夹（而不是空文件夹）中初始化 Git 仓库来进行版本控制，则应该开始跟踪这些文件并提交。可通过 `git add` 命令来实现对指定文件的跟踪，然后执行 `git commit` 提交：

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

现在已经得到了一个实际维护（或者跟踪）着若干文件的 Git 仓库。

(2) 克隆现有的仓库。

如果想获得一份已经存在了的 Git 仓库的副本，比如，想为某个开源项目贡献自己的一份力，这时就要用到 `git clone` 命令。如果对其他的 VCS 系统（比如 Subversion）很熟悉，请留心一下你所使用的命令是 `clone` 而不是 `checkout`。这是 Git 区别于其他版本控制系统的一个重要特性，Git 克隆的是该 Git 仓库服务器上的几乎所有数据，而不是仅仅复制完成工作所需要的文件。当执行 `git clone` 命令的时候，默认配置下远程 Git 仓库中的每一个文件的每一个版本都将被拉取下来。事实上，如果服务器的磁盘坏掉了，则通常可以使用任何一个克隆下来的用户端来重建服务器上的仓库（虽然可能会丢失某些服务器端的挂钩设置，但是所有版本的数据仍在）。

克隆仓库的命令格式是 `git clone [url]`。比如，要克隆 Git 的可链接库 `libgit2`，可使用下面的

命令：

```
$ git clone https://github.com/libgit2/libgit2
```

这会在当前目录下创建一个名为“libgit2”的目录，并在这个目录下初始化一个.git文件夹，从远程仓库拉取下所有数据放入.git文件夹，然后从中读取最新版本文件的副本。进入这个新建的libgit2文件夹，会发现所有的项目文件已经在里面了，准备就绪等待后续的开发和使用。如果想在克隆远程仓库的时候自定义本地仓库的名字，则可以使用如下命令：

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

这将执行与上一个命令相同的操作，在本地创建的仓库名字变为mylibgit。

Git支持多种数据传输协议。上面的例子使用的是https://协议，也可以使用git://协议或SSH传输协议，比如user@server:path/to/repo.git。

2. 记录每次更新到仓库

现在我们有了项目的Git仓库，并从这个仓库中取出了所有文件的工作副本。接下来，对这些文件做些修改，在完成了一个阶段的目标之后，提交本次更新到仓库。

工作目录下的每一个文件都不外乎这两种状态：已跟踪（tracked）或未跟踪（untracked）。已跟踪的文件是指那些被纳入了版本控制的文件，在上一次快照中有它们的记录，在工作一段时间后，它们的状态可能处于已提交（committed）、已修改（modified）或已暂存（staged）。工作目录中除了已跟踪文件，其他所有文件都属于未跟踪文件，它们既不存在于上次快照的记录中，也没有放入暂存区。初次克隆某个仓库的时候，工作目录中的所有文件都属于已跟踪文件，并处于未修改状态。

编辑过某些文件之后，Git将它们标记为已修改文件。我们逐步将这些修改过的文件放入暂存区，然后提交所有暂存的修改，如此反复。所以使用Git时文件的生命周期如图7-17所示。

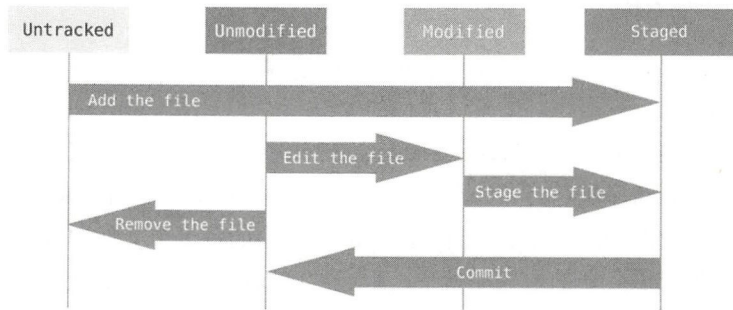


图 7-17 文件的生命周期

检查当前文件状态

要查看哪些文件处于什么状态，可以用 `git status` 命令。如果在克隆仓库后立即使用此命令，则会看到类似这样的输出：

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

这说明现在的工作目录相当干净。换句话说，所有已跟踪文件在上次提交后都未被更改过。此外，上面的信息还表明，当前目录下没有出现任何处于未跟踪状态的新文件，否则 Git 会在这里列出来。最后，该命令还显示了当前所在分支，并告诉你这个分支同远程服务器上对应的分支没有偏离。现在，分支名是“master”，这是默认的分支名。

现在，在项目下创建一个新的 README 文件。如果之前并不存在这个文件，则使用 `git status` 命令后将看到一个新的未跟踪文件：

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

nothing added to commit but untracked files present (use "git add" to track)
```

在状态报告中可以看到新建的 README 文件出现在 Untracked files 下面。未跟踪的文件意味着 Git 在之前的快照（提交）中没有这些文件，Git 不会自动将之纳入跟踪范围，除非你明明白白地告诉它“我需要跟踪该文件”，这样的处理让你不必担心将生成的二进制文件或其他不想被跟踪的文件包含进来。不过在现在的例子中，我们确实想跟踪管理 README 这个文件。

跟踪新文件

使用命令 `git add` 开始跟踪一个文件。要跟踪 README 文件，运行：

```
$ git add README
```

此时再运行 `git status` 命令，会看到 README 文件已被跟踪，并处于暂存状态：

```
$ git status
On branch master
```

```
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
new file:   README
```

Changes to be committed 这行下面的文件就说明是已暂存状态。如果此时提交，那么该文件此时此刻的版本将被留存在历史记录中。之前我们使用 `git init` 后就运行了 `git add (files)` 命令，开始跟踪当前目录下的文件。`git add` 命令使用文件或目录的路径作为参数，如果参数是目录的路径，该命令将递归地跟踪该目录下的所有文件。

暂存已修改文件

现在我们修改一个已被跟踪的文件。如果修改了一个名为 `CONTRIBUTING.md` 的已被跟踪的文件，然后运行 `git status` 命令，则会看到下面的内容：

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

modified:   CONTRIBUTING.md
```

`CONTRIBUTING.md` 文件出现在 `Changes not staged for commit` 这行的下面，说明已跟踪文件的内容发生了变化，但还没有放到暂存区。要暂存这次更新，需要运行 `git add` 命令。这是个多功能命令：可以用它开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等。将这个命令理解为“添加内容到下一次提交中”而不是“将一个文件添加到项目中”要更加合适。现在运行 `git add`，将 `CONTRIBUTING.md` 放到暂存区，然后看看 `git status` 的输出：

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
```



```

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

```

现在两个文件都已暂存，下次提交时就会一并记录到仓库。假设此时想要在 CONTRIBUTING.md 里再加一条注释，重新编辑存盘后，准备好提交。不过且慢，再运行 `git status` 命令看看：

```

$ vim CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md

```

现在 CONTRIBUTING.md 文件同时出现在暂存区和非暂存区。实际上 Git 只不过暂存了运行 `git add` 命令时的版本，如果现在提交，则 CONTRIBUTING.md 的版本是最后一次运行 `git add` 命令时的那个版本，而不是运行 `git commit` 时在工作目录中的当前版本。所以，运行 `git add` 之后又做了修订的文件，需要重新运行 `git add` 命令把最新版本重新暂存起来：

```

$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

```



状态简览

`git status` 命令的输出十分详细，但其用语有些烦琐。如果使用 `git status -s` 命令或 `git status --short` 命令，则得到一种更为紧凑的格式输出：

```
$ git status -s
M README
MM Rakefile
A lib/git.rb
M lib/simplegit.rb
?? LICENSE.txt
```

新添加的未跟踪文件前面有??标记，新添加到暂存区中的文件前面有 A 标记，修改过的文件前面有 M 标记。你可能注意到了 M 有两个可以出现的位置，出现在右边的 M 表示该文件被修改但还没放入暂存区，出现在靠左边的 M 表示该文件被修改并放入了暂存区。例如，上面的状态报告显示 README 文件在工作区被修改但还没有将修改后的文件放入暂存区；lib/simplegit.rb 文件被修改并将修改后的文件放入了暂存区；而 Rakefile 在工作区被修改并提交到暂存区后又在工作区中被修改，所以在暂存区和工作区都有该文件被修改的记录。

忽略文件

一般总会有些文件无须纳入 Git 的管理，也不希望它们总出现在未跟踪文件列表中。通常都是些自动生成的文件，比如日志文件，或者编译过程中创建的临时文件等。在这种情况下，我们可以创建一个名为.gitignore的文件，列出要忽略的文件模式。来看一个实际的例子：

```
$ cat .gitignore
*.o
*.a
*~
```

第一行告诉 Git 忽略所有以.o或.a结尾的文件。一般这类对象文件和存档文件都是在编译过程中出现的。第二行告诉 Git 忽略所有以波浪符(~)结尾的文件，许多文本编辑软件（比如 Emacs）都用这样的文件名来保存副本。此外，可能还需要忽略 log、tmp 或 pid 目录，以及自动生成的文档等。要养成一开始就设置好.gitignore文件的习惯，以免将来误提交这类无用的文件。

文件.gitignore 的格式规范如下：

- 所有空行或者以#开头的行都会被 Git 忽略；
- 可以使用标准的 glob 模式匹配；
- 匹配模式可以以 (/) 开头防止递归；
- 匹配模式可以以 (/) 结尾指定目录；



- 要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号(!)取反。

所谓的 glob 模式是指 shell 所使用的简化的正则表达式。星号(*)匹配零个或多个任意字符；[abc]匹配任何一个列在方括号中的字符(这个例子要么匹配一个 a，要么匹配一个 b，要么匹配一个 c)；问号(?)只匹配一个任意字符；如果在方括号中使用短画线分隔两个字符，则表示所有在这两个字符范围内的都可以匹配(比如[0~9]表示匹配所有 0 到 9 的数字)。使用两个星号(*)表示匹配任意中间目录，比如 a/**/z 可以匹配 a/z、a/b/z 或 a/b/c/z 等。

查看已暂存和未暂存的修改

如果 git status 命令的输出过于模糊，而我们想知道具体修改了什么地方，则可以用 git diff 命令。

假如再次修改 README 文件后暂存，然后编辑 CONTRIBUTING.md 文件后先不暂存，运行 status 命令将会看到：

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

要查看尚未暂存的文件更新了哪些部分，不加参数直接输入 git diff:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@
@@ branch directly, things can get messy.
Please include a nice description of your changes when you submit your PR;
if we have to read the whole diff to figure out why you're contributing
in the first place, you're less likely to get feedback and have your change
merged in.
```



```
+merged in. Also, split your changes into comprehensive chunks if your patch is  
+longer than a dozen lines.
```

If you are starting to work on a particular area, feel free to submit a PR that highlights your work in progress (and note in the PR title that it's...

此命令比较的是工作目录中当前文件和暂存区域快照之间的差异，也就是修改之后还没有暂存起来的变化内容。

若要查看已暂存的将要添加到下次提交里的内容，则可以用 `git diff --cached` 命令（Git 1.6.1 及更高版本还允许使用 `git diff --staged`，效果是相同的，但更好记）。

```
$ git diff --staged  
diff --git a/README b/README  
new file mode 100644  
index 0000000..03902a1  
--- /dev/null  
+++ b/README  
@@ -0,0 +1 @@  
+My Project
```

请注意，`git diff` 本身只显示尚未暂存的改动，而不是自上次提交以来所做的所有改动。所以，有时候暂存所有更新过的文件并运行 `git diff` 后却什么也没有，就是这个原因。

像之前说的，暂存 `CONTRIBUTING.md` 后再编辑，运行 `git status` 会看到暂存前后的两个版本：

```
$ git add CONTRIBUTING.md  
$ echo '# test line' >> CONTRIBUTING.md  
$ git status  
On branch master  
Your branch is up-to-date with 'origin/master'.  
Changes to be committed:  
  (use "git reset HEAD <file>..." to unstage)  
  
    modified:   CONTRIBUTING.md  
  
Changes not staged for commit:  
  (use "git add <file>..." to update what will be committed)  
  (use "git checkout -- <file>..." to discard changes in working directory)  
  
    modified:   CONTRIBUTING.md
```





现在运行 `git diff`，查看暂存前后的变化：

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects
```

```
See our [projects list] (https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
```

```
+# test line
```

然后用 `git diff --cached` 查看已经暂存起来的文件的变化（`--staged` 和 `--cached` 是 synonym）：

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
```

```
Please include a nice description of your changes when you submit your PR;
if we have to read the whole diff to figure out why you're contributing
in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.
```

```
If you are starting to work on a particular area, feel free to submit a PR
that highlights your work in progress (and note in the PR title that it's...
```

提交更新

现在的暂存区域已经准备妥当可以提交了。在此之前，一定要确认还有什么修改过或新建的文件还没有执行 `git add`，否则提交的时候不会记录这些还没暂存起来的变化。这些修改过的文件只保留在本地磁盘。所以每次准备提交前，先用 `git status` 看一下是不是都已暂存起来了，然后运行提交命令 `git commit`：

```
$ git commit
```





这种方式会启动文本编辑器以便输入本次提交的说明。

另外，也可以在 `commit` 命令后添加 `-m` 选项，将提交信息与命令放在同一行：

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
2 files changed, 2 insertions(+)
create mode 100644 README
```

现在已经创建了第一个提交！可以看到，提交后它会告诉你，当前是在哪个分支（`master`）提交的，本次提交的完整 `SHA-1` 校验和是什么（`463dc4f`），以及在本次提交中，有多少文件修订过、多少行添加和删改过。

请记住，提交时记录的是放在暂存区域的快照。任何还未暂存的文件仍然保持已修改状态，可以在下次提交时纳入版本管理。每一次运行提交操作，都是对项目做一次快照，以后可以回到这个状态，或者进行比较。

跳过使用暂存区域

尽管使用暂存区域的方式可以精心准备要提交的细节，但有时候这么做略显烦琐。Git 提供了一个跳过使用暂存区域的方式，只要在提交的时候，给 `git commit` 加上 `-a` 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 `git add` 步骤：

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
1 file changed, 5 insertions(+), 0 deletions(-)
```

这样，提交之前不再需要“`git add`”文件 `CONTRIBUTING.md` 了。

移除文件

要从 Git 中移除某个文件，就必须要从已跟踪文件清单中移除（确切地说，是从暂存区域移除），然后提交。可以用 `git rm` 命令完成此项工作，并连带从工作目录中删除指定的文件，





这样以后就不会出现在未跟踪文件清单中了。

如果只是简单地从工作目录中手工删除文件，则在运行 `git status` 时就会在 “Changes not staged for commit” 部分看到如下内容：

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
    deleted:    PROJECTS.md
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

然后运行 `git rm` 来记录此次移除文件的操作：

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
    deleted:    PROJECTS.md
```

下一次提交时，该文件就不再纳入版本管理了。如果删除之前修改过并且已经放到暂存区域，则必须要用强制删除选项 `-f`。这是一种安全特性，用于防止误删还没有添加到快照的数据，这样的数据不能被 Git 恢复。

另外一种情况是，我们想把文件从 Git 仓库中删除（即从暂存区域移除），但仍然希望保留在当前工作目录中。换句话说，想让文件保留在磁盘，但是并不想让 Git 继续跟踪。当忘记添加 `.gitignore` 文件，不小心把一个很大的日志文件或一堆编译生成文件添加到暂存区时，这一做法尤其有用。为达到这一目的，使用 `--cached` 选项：

```
$ git rm --cached README
```

`git rm` 命令后面可以列出文件或目录的名字，也可以使用 `glob` 模式，例如：

```
$ git rm log/*.log
```





注意到星号(*)之前的反斜杠(\)，因为 Git 有自己的文件模式扩展匹配方式，所以我们不用 shell 来帮忙展开。此命令删除 log/目录下扩展名为.log 的所有文件。类似的：

```
$ git rm *~
```

该命令可删除以~结尾的所有文件。

移动文件

不像其他的 VCS 系统，Git 并不显式跟踪文件移动操作。如果在 Git 中重命名了某个文件，则仓库中存储的元数据并不会体现出这是一次改名操作。不过 Git 非常聪明，它会推断出究竟发生了什么。

既然如此，当你看到 Git 的 mv 命令时一定会困惑不已。要在 Git 中修改文件名，可以这么做：

```
$ git mv file_from file_to
```

它会恰如预期般正常工作。实际上，即便此时查看状态信息，也会看到关于重命名操作的说明：

```
$ git mv README.md README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

其实，运行 git mv 就相当于运行下面三条命令：

```
$ mv README.md README
$ git rm README.md
$ git add README
```

如此分开操作，Git 也会意识到这是一次改名，所以不管何种方式结果都一样。两者唯一的区别是，mv 是一条命令而另一种方式需要三条命令，直接用 git mv 轻便得多。如果想要用其他工具批处理改名，要记得在提交前删除老的文件名，再添加新的文件名。

3. 查看提交历史

在提交若干更新，或者克隆某个项目之后，你也许想回顾一下提交历史，完成这个任务最简单而又有效的工具是 git log 命令。

在项目中运行 git log，应该会看到下面的输出：





```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700
```

```
    changed the version number
```

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700
```

```
    removed unnecessary test
```

```
commit allbef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700
```

```
    first commit
```

一个常用的选项是`-p`，用来显示每次提交的内容差异。也可以加上`-2`来仅显示最近两次提交：

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700
```

```
    changed the version number
```

```
diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 @@ require 'rake/gempackagetask'
spec = Gem::Specification.new do |s|
  s.platform = Gem::Platform::RUBY
  s.name      = "simplegit"
-  s.version  = "0.1.0"
+  s.version  = "0.1.1"
  s.author   = "Scott Chacon"
```





```
s.email      = "schacon@gee-mail.com"
s.summary    = "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
   end

end

-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file
```

如果想看到每次提交的简略的统计信息，则可以使用--stat 选项：

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700
```

```
    changed the version number
```

```
Rakefile | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700
```



```

removed unnecessary test

lib/simplegit.rb | 5 -----
1 file changed, 5 deletions(-)

commit allbef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sat Mar 15 10:31:28 2008 -0700

first commit

README           | 6 ++++++
Rakefile         | 23 ++++++++++++++++++++++
lib/simplegit.rb | 25 ++++++++++++++++++++++
3 files changed, 54 insertions(+)

```

另外一个常用的选项是`--pretty`。这个选项可以指定使用不同于默认格式的方式展示提交历史，并且有一些内建的子选项可供使用。比如，用 `oneline` 将每个提交放在一行显示，在查看的提交数很大时非常有用。另外还有 `short`、`full` 和 `fuller` 可以用，展示的信息或多或少有些不同：

```

$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
allbef06a3f659402fe7563abf99ad00de2209e6 first commit

```

`format` 可以定制要显示的记录格式：

```

$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
allbef0 - Scott Chacon, 6 years ago : first commit

```

当 `oneline` 或 `format` 与另一个 `log` 选项`--graph` 结合使用时尤其有用。这个选项添加了一些 ASCII 字符串来形象地展示分支、合并历史：

```

$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.

```



```
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
* 11d191e Merge branch 'defunkt' into local*
```

限制输出长度

按照时间进行限制的选项，比如--since 和--until 也很有用。例如，下面的命令列出了所有最近两周内的提交：

```
$ git log --since=2.weeks
```

这个命令可以在多种格式下工作，比如具体的某一天“2008-01-15”，或者相对多久以前“2 years 1 day 3 minutes ago”。

还可以给出若干搜索条件，列出符合的提交。用--author 选项显示指定作者的提交，用--grep 选项搜索提交说明中的关键字（注意，如果要得到同时满足这两个选项搜索条件的提交，则必须用--all-match 选项。否则，满足任意一个条件的提交都会被匹配出来）。

另一个非常有用的筛选选项是-S，可以列出那些添加或移除某些字符串的提交。比如，想找出添加或移除某一个特定函数的引用的提交，可以这样使用：

```
$ git log -Sfunction_name
```

4. 撤销操作

有时候我们提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。此时可以运行带有--amend 选项的提交命令尝试重新提交：

```
$ git commit -amend
```

这个命令会将暂存区中的文件提交。如果自上次提交以来还未做任何修改（例如，在上次提交后马上执行了此命令），那么快照会保持不变，而修改的只是提交信息。

文本编辑器启动后，可以看到之前的提交信息。编辑后保存会覆盖原来的提交信息。

例如，提交后发现忘记暂存某些需要的修改，可以像下面这样操作：

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit -amend
```

最终只会有一个提交。第二次提交将代替第一次提交的结果。



取消暂存的文件

已经修改两个文件并且想要将它们作为两次独立的修改提交，但是却意外地输入 `git add *` 暂存了它们两个。如何只取消暂存两个中的一个呢？`git status` 命令提示：

```
$ git add *
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

在“Changes to be committed”文字正下方，提示使用 `git reset HEAD <file>...` 来取消暂存。所以，我们可以这样来取消暂存 `CONTRIBUTING.md` 文件：

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M  CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

`CONTRIBUTING.md` 文件已经是修改未暂存的状态了。

撤销对文件的修改

如果并不想保留对 `CONTRIBUTING.md` 文件的修改该怎么办？如何方便地撤销修改，将它还原成上次提交时的样子（或者刚克隆完的样子，或者刚把它放入工作目录时的样子）呢？幸运的是，`git status` 也告诉了我们应该如何做。在最后一个例子中，未暂存区域是这样的：

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
```



```
(use "git checkout -- <file>..." to discard changes in working directory)
```

```
modified:   CONTRIBUTING.md
```

它非常清楚地告诉了我们如何撤销之前所做的修改。让我们来按照提示执行：

```
$ git checkout -- CONTRIBUTING.md
```

```
$ git status
```

```
On branch master
```

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

```
renamed:    README.md -> README
```

可以看到那些修改已经被撤销了。

5. 远程仓库的使用

远程仓库是指托管在因特网或其他网络中项目的版本库。管理远程仓库包括了解如何添加远程仓库，移除无效的远程仓库，管理不同的远程分支并定义它们是否被跟踪，等等。

查看远程仓库

如果想查看已经配置的远程仓库服务器，则可以运行 `git remote` 命令。它会列出我们指定的每一个远程服务器的简写：

```
$ git clone https://github.com/schacon/ticgit
```

```
Cloning into 'ticgit'...
```

```
remote: Reusing existing pack: 1857, done.
```

```
remote: Total 1857 (delta 0), reused 0 (delta 0)
```

```
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
```

```
Resolving deltas: 100% (772/772), done.
```

```
Checking connectivity... done.
```

```
$ cd ticgit
```

```
$ git remote
```

```
origin
```

也可以指定选项 `-v`，会显示需要读写远程仓库使用的 Git 保存的简写与其对应的 URL。

```
$ git remote -v
```

```
origin https://github.com/schacon/ticgit (fetch)
```

```
origin https://github.com/schacon/ticgit (push)
```

如果远程仓库不止一个，则该命令会将它们全部列出。例如，与几个协作者合作的，拥有

多个远程仓库的仓库看起来像下面这样：

```
$ cd grit
$ git remote -v
bakkdoor https://github.com/bakkdoor/grit (fetch)
bakkdoor https://github.com/bakkdoor/grit (push)
cho45 https://github.com/cho45/grit (fetch)
cho45 https://github.com/cho45/grit (push)
defunkt https://github.com/defunkt/grit (fetch)
defunkt https://github.com/defunkt/grit (push)
koke git://github.com/koke/grit.git (fetch)
koke git://github.com/koke/grit.git (push)
origin git@github.com:mojombo/grit.git (fetch)
origin git@github.com:mojombo/grit.git (push)
```

这样我们可以轻松拉取其中任何一个用户的贡献。

添加远程仓库

运行 `git remote add <shortname> <url>` 添加一个新的远程 Git 仓库，同时指定一个你可以轻松引用的简写：

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
pb https://github.com/paulboone/ticgit (fetch)
pb https://github.com/paulboone/ticgit (push)
```

现在可以在命令行中使用字符串 `pb` 来代替整个 URL。例如，如果想拉取 Paul 仓库中有但你没有的信息，则可以运行 `git fetch pb`：

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
* [new branch]      master      -> pb/master
* [new branch]      ticgit      -> pb/ticgit
```

现在 Paul 的 master 分支可以在本地通过 pb/master 访问到。你可以将它合并到自己的某个分支中，如果想要查看它，则可以检出一个指向该点的本地分支。

从远程仓库中抓取与拉取

就如刚才所见，从远程仓库中获得数据，可以执行：

```
$ git fetch [remote-name]
```

这个命令会访问远程仓库，从中拉取所有你还没有的数据。执行完成后，你将拥有那个远程仓库中所有分支的引用，可以随时合并或查看。

如果使用 clone 命令克隆了一个仓库，则命令会自动将其添加为远程仓库并默认“origin”为简写。所以，git fetch origin 会抓取克隆（或上一次抓取）后新推送的所有工作。必须注意 git fetch 命令会将数据拉取到你的本地仓库中。它并不会自动合并或修改你当前的工作。

如果有一个分支设置为跟踪一个远程分支，则可以使用 git pull 命令来自动地抓取，然后合并远程分支到当前分支。这对你来说可能是一个更简单或更舒服的工作流程；默认情况下，git clone 命令会自动设置本地 master 分支跟踪克隆的远程仓库的 master 分支（或者不管是什么名字的默认分支）。运行 git pull 通常会从最初克隆的服务器上抓取数据并自动尝试合并到当前所在的分支。

推送到远程仓库

当你想分享项目时，必须将其推送到上游。这个命令很简单：git push [remote-name] [branch-name]。当想要将 master 分支推送到 origin 服务器时，那么运行这个命令就可以将你所做的备份到服务器：

```
$ git push origin master
```

只有当你有所克隆服务器的写入权限，并且之前没有人推送过时，这条命令才能生效。当你和其他人在同一时间克隆，他们先推送到上游，然后你推送到上游时，你的推送就会毫无疑问地被拒绝。你必须先将他们的工作拉取下来并将其合并进你的工作后才能推送。

查看远程仓库

如果想要查看某一个远程仓库的更多信息，则可以使用 git remote show [remote-name] 命令。如果想以一个特定的缩写名运行这个命令，例如 origin，则会得到像下面这样的信息：

```
$ git remote show origin
* remote origin
Fetch URL: https://github.com/schacon/ticgit
Push URL: https://github.com/schacon/ticgit
HEAD branch: master
Remote branches:
```



```

master                                tracked
dev-branch                            tracked
Local branch configured for 'git pull':
  master merges with remote master
Local ref configured for 'git push':
  master pushes to master (up to date)

```

它同样会列出远程仓库的 URL 与跟踪分支的信息。这些信息非常有用，它告诉你正处于 `master` 分支，并且如果运行 `git pull`，则会抓取所有的远程引用，然后将远程 `master` 分支合并到本地 `master` 分支。它也会列出拉取的所有远程引用。

这是一个经常遇到的简单例子。如果你是 Git 的重度使用者，那么还可以通过 `git remote show` 看到更多的信息。

```

$ git remote show origin
* remote origin
URL: https://github.com/my-org/complex-project
Fetch URL: https://github.com/my-org/complex-project
Push URL: https://github.com/my-org/complex-project
HEAD branch: master
Remote branches:
  master                                tracked
  dev-branch                            tracked
  markdown-strip                        tracked
  issue-43                             new (next fetch will store in remotes/origin)
  issue-45                             new (next fetch will store in remotes/origin)
  refs/remotes/origin/issue-11         stale (use 'git remote prune' to remove)
Local branches configured for 'git pull':
  dev-branch merges with remote dev-branch
  master      merges with remote master
Local refs configured for 'git push':
  dev-branch      pushes to dev-branch      (up to date)
  markdown-strip  pushes to markdown-strip  (up to date)
  master          pushes to master          (up to date)

```

这个命令列出了当你在特定的分支上执行 `git push` 时会自动地推送到哪一个远程分支。它同样列出了哪些远程分支不在你的本地，哪些远程分支已经从服务器上移除了，还有当执行 `git pull` 时哪些分支会自动合并。

远程仓库的移除与重命名

如果想要重命名引用的名字，则可以运行 `git remote rename` 去修改一个远程仓库的简写名。例如，想要将 `pb` 重命名为 `paul`，可以用 `git remote rename` 这样做：

```
$ git remote rename pb paul
$ git remote
origin
paul
```

值得注意的是，这同样会修改远程分支的名字。那些过去引用 `pb/master` 的现在会引用 `paul/master`。

如果因为一些原因想要移除一个远程仓库——已经从服务器上搬走了或不再使用某一个特定的镜像，或者某一个贡献者不再提交贡献了，可以使用 `git remote rm` 命令：

```
$ git remote rm paul
$ git remote
origin
```

6. 打标签

像其他版本控制系统（VCS）一样，Git 可以给历史中的某一个提交打上标签，以示重要。比较有代表性的是，人们会使用这个功能来标记发布结点（v1.0 等）。

列出标签

在 Git 中列出已有的标签是非常简单直观的。只需要输入 `git tag`：

```
$ git tag
v0.1
v1.3
```

这个命令以字母顺序列出标签。

也可以使用特定的模式查找标签。例如，Git 自身的源代码仓库包含标签的数量超过 500 个。如果只对 1.8.5 系列感兴趣，则可以运行：

```
$ git tag -l "v1.8.5*"
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
```

```
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

创建标签

Git 使用两种主要类型的标签：轻量标签（lightweight）与附注标签（annotated）。

一个轻量标签很像一个不会改变的分支，它只是一个特定提交的引用。

然而，附注标签是存储在 Git 数据库中的一个完整对象。它们是可以被校验的，其中包含打标签者的名字、电子邮件地址、日期时间，有一个标签信息，并且可以使用 GNU Privacy Guard（GPG）签名与验证。通常建议创建附注标签，这样可以拥有以上所有信息；如果只是想用—个临时的标签，或者因为某些原因不想要保存那些信息，则轻量标签也是可用的。

附注标签

在 Git 中创建一个附注标签是很简单的。最简单的方式是在运行 tag 命令时指定 -a 选项：

```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
v0.1
v1.3
v1.4
```

-m 选项指定了一条将会存储在标签中的信息。如果没有为附注标签指定一条信息，则 Git 会运行编辑器要求你输入信息。

通过使用 git show 命令可以看到标签信息与对应的提交信息：

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date: Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number
```

输出显示了打标签者的信息、打标签的日期时间、附注信息，然后显示具体的提交信息。

轻量标签

另一种给提交打标签的方式是使用轻量标签。轻量标签本质上是提交校验和存储到一个文件中，不会保存任何其他信息。

创建轻量标签，不需要使用 `-a`、`-s` 或 `-m` 选项，只需要提供标签名字：

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

这时，如果在标签上运行 `git show`，则不会看到额外的标签信息。命令只会显示出提交信息：

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700
```

```
changed the version number
```

后期打标签

也可以对过去的提交打标签。假设提交历史是这样的：

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbb added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

现在，假设在 `v1.2` 时忘记给项目打标签，也就是在“`updated rakefile`”时提交，那么可以在之后补上标签，即在命令的末尾指定提交的校验和（或部分校验和）：

```
$ git tag -a v1.2 9fceb02
```


可以看到已经在那次提交上补上标签了：

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
```

共享标签

默认情况下，`git push` 命令并不会传送标签到远程仓库服务器上。在创建完标签后必须显式地将标签推送到共享服务器上。这个过程就像共享远程分支一样。可以运行 `git push origin [tagname]`：

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]          v1.5 -> v1.5
```

如果想要一次性推送很多标签，则可以使用带有 `--tags` 选项的 `git push` 命令，这将会把所有不在远程仓库服务器上的标签全部传送到那里。

```
$ git push origin --tags
```

```
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
* [new tag]          v1.4 -> v1.4
* [new tag]          v1.4-lw -> v1.4-lw
```

现在，当其他人从仓库中克隆或拉取时，他们也能得到那些标签。

检出标签

在 Git 中并不能真的检出一个标签，因为它们并不能像分支一样来回移动。如果想要工作目录与仓库中特定的标签版本完全一样，则可以使用 `git checkout -b [branchname] [tagname]` 在特定的标签上创建一个新分支：

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

当然，如果在这之后又进行了一次提交，`version2` 分支会因为改动向前移动了，那么 `version2` 分支就会和 `v2.0.0` 标签稍微有些不同，这时就应该当心了。

7. Git 别名

别名可以使 Git 的体验更简单、容易、熟悉。

如果不想每次都输入完整的 Git 命令，则可以通过 `git config` 文件来轻松地为一个命令设置一个别名。这里有一些例子：

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

这意味着，当要输入 `git commit` 时，只需要输入 `git ci`。

在创建你认为应该存在的命令时这个技术会很有用。例如，为了解决取消暂存文件的易用性问题，可以向 Git 中添加你自己的取消暂存别名：

```
$ git config --global alias.unstage 'reset HEAD --'
```

这会使下面的两个命令等价：

```
$ git unstage fileA
$ git reset HEAD - fileA
```

这样看起来更清楚一些。通常也会添加一个 `last` 命令，像这样：

```
$ git config --global alias.last 'log -1 HEAD'
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date: Tue Aug 26 19:48:51 2008 +0800
```

```
test for current head
```

```
Signed-off-by: Scott Chacon schacon@example.com
```

可以看出, Git 只是简单地将别名替换为对应的命令。然而, 你可能想要执行外部命令, 而不是一个 Git 子命令。如果是那样, 则可以在命令前面加入!符号。如果要写一些与 Git 仓库协作的工具, 则会很有用。我们现在将 `git visual` 定义为 `gitk` 的别名:

```
$ git config --global alias.visual '!gitk'
```

7.5 Git Flow——团队协作最佳实践

由于 Git 的使用本身是非常灵活的, 如果滥用 Git, 不但不利于版本的管理, 甚至会造成管理上的混乱。Git 被大型互联网公司广泛采用, 在使用 Git 的过程中, 每个公司都根据自己的实际情况指定了使用 Git 的规范。这些使用规范有利于团队成员之间降低沟通成本, 使团队朝着一致的管理目标前进。

在众多使用规范中, Git Flow 是公认的使用 Git 进行团队协作的最佳实践。顾名思义, Git Flow 就是定义了一套使用 Git 的流程。

当然, 你也可以安装一套 `git-flow`, 这样, 你将会拥有一些扩展命令。这些命令会在一个预定义的顺序下自动执行多个操作, 而这些操作就是我们想要的工作流程。`git-flow` 并不是要替代 Git, 它仅仅是非常聪明有效地把标准的 Git 命令用脚本组合了起来。所以从这个角度来说, 即便不安装 `git-flow`, 仍然可以使用 Git Flow 所定义的工作流程。你只需要了解哪些工作流程是由哪些单独的任务所组成的, 并且附上正确的参数, 以及在一个正确的顺序下简单执行那些对应的 Git 命令就可以了。当然, 使用 `git-flow` 所带来的好处在于, 不需要把这些命令和顺序都记在脑子里。

7.5.1 分支定义

Git Flow 对于分支的命名有严格定义。比如:

- **master**——只能用来包括产品代码。一般开发人员不能直接工作在这个 `master` 分支上,

而是在其他指定的、独立的特性分支中。

- **develop**——进行任何新的开发的基础分支。当开始一个新的功能分支时，它将是开发的基础。另外，该分支也汇集所有已经完成的功能，并等待被整合到 **master** 分支中。
- **feature**——基于 **develop** 分支所检出的用于开发特性功能的分支。
- **hotfix**——基于产品代码（一般是指 **master** 分支）检出的用于修复产品 Bug 的分支。
- **release**——基于 **develop** 分支所检出的用于发布版本的分支。

7.5.2 新功能开发 workflow

对于一个开发人员来说，最平常的工作可能就是功能的开发。让我们一起看一下 **git-flow** 是如何定义功能开发工作流程的。

1. 开始新功能

开发一个新功能“**rss-feed**”：

```
$ git flow feature start rss-feed
Switched to a new branch 'feature/rss-feed'
```

Summary of actions:

- A new branch 'feature/rss-feed' was created, based on 'develop'
- You are now on branch 'feature/rss-feed'

执行上述命令之后，**git-flow** 会创建一个名为“**feature/rss-feed**”的分支。其中，这个“**feature/**”前缀代表我们的分支是一个特性功能的开发。

2. 完成一个功能

经过一段时间工作和一系列的提交，我们的新功能终于完成了：

```
$ git flow feature finish rss-feed
Switched to branch 'develop'
Updating 6bcf266..41748ad
Fast-forward
   feed.xml | 0
   1 file changed, 0 insertions(+), 0 deletions(-)
   create mode 100644 feed.xml
Deleted branch feature/rss-feed (was 41748ad).
```

其中，这个“**feature finish**”命令会把工作整合到主“**develop**”分支中。在这里它需要等待。

之后，git-flow 会进行清理操作。它会删除这个当下已经完成的功能分支，并且换到“develop”分支。

7.5.3 Bug 修复 workflow

对于开发工作而言，Bug 的产生不可避免。我们唯一可以做的就是尽量早地发现 Bug，以及尽快地修复 Bug。

在这种情况下，git-flow 提供一个特定的“hotfix”工作流程，用于修复产品的 Bug。

1. 创建 hotfix

执行下面的命令：

```
$ git flow hotfix start missing-link
```

这个命令会创建一个名为“hotfix/missing-link”的分支。其中，“hotfix/”前缀代表分支是一个产品 Bug 的修复，所以 hotfix 分支基于“master”分支。

这也是和 release 分支最明显的区别，release 分支都是基于“develop”分支的。因为不应该在一个还不完全稳定的开发分支上对产品代码进行修复。

2. 完成 hotfix

在把修复提交到 hotfix 分支之后，就该去完成它了：

```
$ git flow hotfix finish missing-link
```

这个过程非常类似于发布一个 release 版本：

完成的改动会被合并到“master”中，同样会合并到“develop”分支中，这样就可以确保这个错误不会再次出现在下一个 release 中。

随后，hotfix 分支将被删除，然后切换到“develop”分支上去。

7.5.4 版本发布 workflow

当功能开发完成，并且通过了测试，那么此时就具备了版本发布的条件。让我们来看看如何利用 git-flow 创建和发布 release。

1. 创建 release

当在“develop”分支中的代码已经是一个成熟的 release 版本时，这意味着：

- 它包括所有新的功能和必要的修复；

- 它已经被彻底地测试过了。

如果上述两点都满足，则是时候开始生成一个新的 release 了：

```
$ git flow release start 1.1.5  
Switched to a new branch 'release/1.1.5'
```

注意，release 分支是使用版本号命名的。这是一个明智的选择，这个命名方案还有一个很好的附带功能，那就是当我们完成 release 后，git-flow 会适当地自动去标记那些 release 提交。

2. 完成 release

执行下面的命令来完成 release：

```
git flow release finish 1.1.5
```

这个命令会完成如下一系列的操作：

首先，git-flow 会拉取远程仓库，以确保目前是最新的版本。

然后，release 的内容会被合并到“master”和“develop”两个分支中，这样不仅产品代码为最新的版本，而且新的功能分支也将基于最新代码。

为便于识别和做历史参考，release 提交会被标记上这个 release 的名字（在我们的例子里是“1.1.5”）。

清理操作，版本分支会被删除，并且回到“develop”。



第 8 章

RESTful API、微服务及容器技术

8.1 Jersey

Jersey 是基于 Java 的一个轻量级 RESTful 风格的 Web 服务框架。有关 Jersey 的内容，已经在 2.3 节“REST 风格的架构”中做了简单的介绍。本节将详细介绍 Jersey 的使用方法。

8.1.1 Jersey 简介

开发 RESTful Web 服务并不是一件容易的事，不但需要无缝支持在各种媒体类型表示中展示数据，而且还要对客户端-服务器通信的底层细节进行封装抽象。为了简化 Java 在 RESTful Web 服务及其客户端中的开发，目前已经设计了一个标准、便携的 JAX-RS。Jersey 就是用于在 Java 中开发 RESTful Web 服务的开源且适用于生产环境的框架，提供对 JAX-RS 的支持，并用于 JAX-RS（JSR 311 和 JSR 339）的参考实现。

Jersey 框架比 JAX-RS 参考实现得更多。Jersey 提供了自己的 API，扩展了 JAX-RS 工具包及其他功能，以进一步简化 RESTful 的服务和客户端开发。Jersey 还公开了大量的扩展 SPI，以便开发人员可以扩展 Jersey 以适合他们的需要。

Jersey 项目的目标可以总结为以下几点：

- 作为 GlassFish 子项目的 JAX-RS 的参考实现，跟踪并定期发布符合生产质量的 API；
- 提供 API 以扩展 Jersey，构建用户和开发人员的社区；
- 使用 Java 和 Java 虚拟机轻松构建 RESTful Web 服务。

8.1.2 Jersey 的模块和依赖

1. 与 Java SE 的兼容性

Jersey 2.6 以前的版本是由 Java SE 6 编译的，2.7 版本后发生了变化。现在几乎所有的 Jersey 组件都用 Java SE 7 目标编译。这意味着，如果要使用最新的 Jersey，则至少需要 Java SE 7 才能够编译并运行应用程序。只有 core-common 和 core-client 模块仍然需要 Java SE 6 编译。

2. Jersey 的依赖

Jersey 的创建、组装和安装都使用 Apache Maven，非快照的 Jersey 都部署到了 Maven 中央库（<http://search.maven.org/>），也部署到了 Java.Net Maven repositories（<http://maven.java.net/>），包括带有快照的版本。当然如果要查看最新的版本也可以从 Java.Net Maven repositories 检出。

一个使用 Jersey 的应用，依赖于 Jersey 的模块。如果使用第三方模块，那么 Jersey 可能反过来依赖第三方模块。Jersey 是插件化的组件结构，所以不同的应用可能依赖不同的模块。

3. 常见的 Jersey 用例

基于 Servlet 的 GlassFish 应用

如果使用 GlassFish 应用服务，那么不需要打包任何东西，一切都已经包含其中了。只需要在应用中声明依赖使用 JAX-RS 即可。

```
<dependency>
  <groupId>javax.ws.rs</groupId>
  <artifactId>javax.ws.rs-api</artifactId>
  <version>2.0.1</version>
  <scope>provided</scope>
</dependency>
```

如果使用特定的功能，那么直接使用特定的 Jersey 依赖即可：

```
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
  <artifactId>jersey-container-servlet</artifactId>
  <version>2.23.2</version>
  <scope>provided</scope>
</dependency>
<!-- 仅使用 JAX-RS Client 时添加 -->
<dependency>
  <groupId>org.glassfish.jersey.core</groupId>
  <artifactId>jersey-client</artifactId>
  <version>2.23.2</version>
  <scope>provided</scope>
</dependency>
```

基于 Servlet 的服务端应用

以下依赖可以应用于没有集成任何 JAX-RS 实现的应用服务器（Servlet 容器），需要在部署的应用里面包含 JAX-RS 和 Jersey 的实现。

```
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
  <!--如果容器实现使用的是 Servlet API 3.0 以前的版本，那么使用“jersey-container-servlet-core”-->
```

```

        <artifactId>jersey-container-servlet</artifactId>
        <version>2.23.2</version>
    </dependency>
    <!-- 仅使用 JAX-RS Client 时添加 -->
    <dependency>
        <groupId>org.glassfish.jersey.core</groupId>
        <artifactId>jersey-client</artifactId>
        <version>2.23.2</version>
    </dependency>

```

运行于 JDK 的客户端应用

在 JDK 上运行的应用，是否使用 JAX-RS 中客户端的规范完全取决于用户。有各种不同的附加模块可以被添加，例如 grizzly、Apache 或 Jetty 等连接器（见下面的依赖）。Jersey 客户端在 JDK 上默认运行 `URLConnection`。更多的细节参见 <https://jersey.java.net/documentation/latest/client.html>。

```

<dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>2.23.2</version>
</dependency>

```

目前可用的连接器有：

```

<dependency>
    <groupId>org.glassfish.jersey.connectors</groupId>
    <artifactId>jersey-grizzly-connector</artifactId>
    <version>2.23.2</version>
</dependency>

```

```

<dependency>
    <groupId>org.glassfish.jersey.connectors</groupId>
    <artifactId>jersey-apache-connector</artifactId>
    <version>2.23.2</version>
</dependency>

```

```

<dependency>
    <groupId>org.glassfish.jersey.connectors</groupId>
    <artifactId>jersey-jetty-connector</artifactId>
    <version>2.23.2</version>

```

```
</dependency>
```

服务器端应用支持的容器

除了标准的 JAX-RS 基于 Servlet 的部署（Servlet 2.5 及以上版本），Jersey 对下面的容器提供了可编程的部署环境：Grizzly 2（HTTP 和 Servlet）、JDK HTTP 服务器、简单的 HTTP 服务器、Jetty HTTP 服务器：

```
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
  <artifactId>jersey-container-grizzly2-http</artifactId>
  <version>2.23.2</version>
</dependency>
```

```
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
  <artifactId>jersey-container-grizzly2-servlet</artifactId>
  <version>2.23.2</version>
</dependency>
```

```
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
  <artifactId>jersey-container-jdk-http</artifactId>
  <version>2.23.2</version>
</dependency>
```

```
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
  <artifactId>jersey-container-simple-http</artifactId>
  <version>2.23.2</version>
</dependency>
```

```
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
  <artifactId>jersey-container-jetty-http</artifactId>
  <version>2.23.2</version>
</dependency>
```

```
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
```

```
<artifactId>jersey-container-jetty-servlet</artifactId>
<version>2.23.2</version>
</dependency>
```

8.1.3 JAX-RS 核心概念

Java API for RESTful Web Services (JAX-RS) 是用于指导在 Java 中开发 RESTful Web 服务的规范。详细的规范细节可以参考在线文档 <https://jax-rs-spec.java.net/>。

目前，JAX-RS 最新的规范为 2.0，Java 规范提案为 JSR 339。下面介绍 JAX-RS 的核心概念。

1. Root Resource Classes（根资源类）

Root Resource Classes 是带有 `@Path` 注解的，包含至少一个 `@Path` 注解的方法，或者方法带有 `@GET`、`@POST`、`@DELETE` 资源方法指示器的 POJO。资源方法是带有资源方法指示器（resource method designator）注解的方法。这一节展示如何使用 Java 对象内的注解来创建一个 Jersey 的 RESTful 服务。

下面这段代码就是一个带有 JAX-RS 注解的简单示例：

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("/helloworld")
public class HelloWorldResource {
    public static final String CLICHED_MESSAGE = "Hello World!";

    @GET
    @Produces("text/plain")
    public String getHello() {
        return CLICHED_MESSAGE;
    }
}
```

下面看一下 JAX-RS 里面的几个常用注解。

`@Path`

`@Path` 是一个 URI 的相对路径，在上面的例子中，设置的是本地的 URI 的 `/helloworld`。这是一个非常简单的关于 `@Path` 的例子，还可以将变量嵌入 URI。

URI 的路径模板是由 URI 和嵌入 URI 语法的变量组成的。变量在运行时将会被匹配到的 URI 的那部分代替。例如，下面的@Path注解：

```
@Path("/users/{username}")
```

按照这种类型的例子，一个用户会方便地填写他的名字，那么 Jersey 服务器也会按照 URI 路径模板响应这个请求。例如，用户输入了名字“Galileo”，服务器就会响应 `http://example.com/users/Galileo`。

为了接收用户名变量，@PathParam 用在接收请求的方法的参数上，例如：

```
@Path("/users/{username}")
public class UserResource {

    @GET
    @Produces("text/xml")
    public String getUser(@PathParam("username") String userName) {
        ...
    }
}
```

它规定匹配正则表达式要精确到字母的大小写，如果填写，则会覆盖默认的表达式 `[^]+?`，例如：

```
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]*}")
```

这个正则表达式匹配由大小写字母、横杠和数字组成的字符串，如果正则校验不通过，则返回 404（没有找到资源）。

一个@Path 的内容是否以“/”开头没有区别，同样是否以“/”结尾也没有区别。

HTTP 方法：@GET、@PUT、@POST、@DELETE 等

@GET、@PUT、@POST、@DELETE、@HEAD 是 JAX-RS 定义的注解，非常类似于 HTTP 的方法名。在上面的例子中，这些注解是通过 HTTP 的 GET 方法实现的。资源的响应就是 HTTP 的响应。

下面这个例子是存储服务的一个片段，使用 PUT 方法处理创建或修改存储容器：

```
@PUT
public Response putContainer() {
    System.out.println("PUT CONTAINER " + container);

    URI uri = uriInfo.getAbsolutePath();
    Container c = new Container(container, uri.toString());
```

```

    Response r;
    if (!MemoryStore.MS.hasContainer(c)) {
        r = Response.created(uri).build();
    } else {
        r = Response.noContent().build();
    }

    MemoryStore.MS.createContainer(c);
    return r;
}

```

如果没有明确的定义，则 JAX-RS 运行的时候默认支持 HEAD 和 OPTIONS 方法。HEAD 运行时将调用 GET 方法的实现（如果存在）和忽略响应实体（如果设置）。一个响应返回 OPTIONS 的方法取决于所要求的媒体类型在头文件中“Accept”的定义。OPTIONS 方法可以返回一组在“Allow”头中支持的资源方法，或者返回 WADL 文档。

@Produces

@Produces 是定义返回值给客户端的 MIME 媒体类型。在下面这个例子中，将会返回一个对应于 text/plain 的 MIME 媒体类型。@Produces 既可以应用在类上，也可以应用在方法上。

```

@Path("/myResource")
@Produces("text/plain")
public class SomeResource {

    @GET
    public String doGetAsPlainText() {
        ...
    }

    @GET
    @Produces("text/html")
    public String doGetAsHtml() {
        ...
    }
}

```

doGetAsPlainText 方法默认使用类水平的 @Produces 注解内容，也就是 text/plain。而 doGetAsHtml 方法使用方法水平的 @Produces，也就是 text/html。也就是说，方法水平层面的 @Produces 会覆盖类层面的 @Produces。

如果一个资源类能够生产多个 MIME 媒体类型，则资源的方法将会响应给客户端对其来说最可接受的媒体类型。由 HTTP 请求头来设置什么是最容易被接受的。例如，如果接受头部是 `Accept: text/plain`，则 `doGetAsPlainText()` 方法会被调用。如果接受标题是 `Accept: text/plain;q=0.9, text/html`，即客户可以接受 `text/plain` 和 `text/html`，但更容易接受后者的媒体类型，则 `doGetAsHtml()` 方法会被调用。

`@Produces` 可以定义多个返回类型，例如：

```
@GET
@Produces({"application/xml", "application/json"})
public String doGetAsXmlOrJson() {
    ...
}
```

无论 `application/xml` 还是 `application/json` 匹配了，都会执行 `doGetAsXmlOrJson`，如果两个都匹配了，那么会选择首先匹配的那个。

服务器也可选择指定个别媒体类型的品质因数，这些是由客户端来决定的。例如：

```
@GET
@Produces({"application/xml; qs=0.9", "application/json"})
public String doGetAsXmlOrJson() {
    ...
}
```

在上面的示例中，如果客户端接受 `application/xml` 或 `application/json`，那么服务器总是发送 `application/json`，因为 `application/xml` 有一个较低的品质因数。

`@Consumes`

`@Consumes` 注释用来指定表示可由资源消耗的 MIME 媒体类型。上面的例子可以修改设置如下：

```
@POST
@Consumes("text/plain")
public void postClickedMessage(String message) {
    ...
}
```

在这个例子中，该 Java 方法将消耗表示确定的 MIME 媒体类型 `text/plain`。注意，资源的方法返回 `void`。这意味着没有内容返回，而是一个 204 状态码响应（204 指“无内容”）将返回客户端。

`@Consumes` 既可以应用在类的水平上，也可以作用于方法的水平上，而且声明可以不止一





种类型。

2. 参数注解 (@*Param)

在资源方法中，带有基于参数注解的参数可以从请求中获取信息。前面的一个例子就是在匹配@Path之后，通过@PathParam来获取URL请求中的路径参数的。

@QueryParam用于从请求URL的查询组件中提取查询参数。观察下面的例子：

```
@Path("smooth")
@GET
public Response smooth(
    @DefaultValue("2") @QueryParam("step") int step,
    @DefaultValue("true") @QueryParam("min-m") boolean hasMin,
    @DefaultValue("true") @QueryParam("max-m") boolean hasMax,
    @DefaultValue("true") @QueryParam("last-m") boolean hasLast,
    @DefaultValue("blue") @QueryParam("min-color") ColorParam minColor,
    @DefaultValue("green") @QueryParam("max-color") ColorParam maxColor,
    @DefaultValue("red") @QueryParam("last-color") ColorParam lastColor) {
    ...
}
```

如果step的参数存在，那么赋值给它，否则默认@DefaultValue定义的值是2。如果step的内容不是32位的整型数，那么会返回404错误。

例如，用户定义了一个Java类型的ColorParam，实现如下：

```
public class ColorParam extends Color {

    public ColorParam(String s) {
        super(getRGB(s));
    }

    private static int getRGB(String s) {
        if (s.charAt(0) == '#') {
            try {
                Color c = Color.decode("0x" + s.substring(1));
                return c.getRGB();
            } catch (NumberFormatException e) {
                throw new WebApplicationException(400);
            }
        } else {
```





```
try {
    Field f = Color.class.getField(s);
    return ((Color)f.get(null)).getRGB();
} catch (Exception e) {
    throw new WebApplicationException(400);
}
}
```

一般地，Java 方法的参数类型可能是：

- 一个原始类型；
- 一个接收字符串参数的构造函数；
- 一个静态方法或一个命名为 `fromString` 的方法，用于接收字符串参数，例如 `Integer.valueOf(String)` 和 `java.util.UUID.fromString(String)`；
- 一个 `javax.ws.rs.ext.ParamConverterProvider` 的 JAX-RS 扩展 SPI 的注册实现，将返回 `javax.ws.rs.ext.ParamConverter` 的实例，用于将字符串转化为指定类型；
- 当参数是同类型的集合时，将会是 `List<T>`、`Set<T>` 或 `SortedSet<T>`，这样的集合是只读的。

有时参数可以包含相同名称的多个值。如果是这样，则上面第 5 种参数类型可以用来获得所有的值。

如果 `@DefaultValue` 不与 `@QueryParam` 联合使用，如果在请求中不存在查询参数，则 `List`、`Set` 或 `SortedSet` 类型将会是空值集合，对象类型将为空，Java 的定义默认为原始类型。

`@PathParam` 和其他参数注解 `@MatrixParam`、`@HeaderParam`、`@CookieParam`、`@FormParam` 遵循与 `@QueryParam` 一样的规则。`@MatrixParam` 从 URL 路径提取信息；`@HeaderParam` 从 HTTP 头部提取信息；`@CookieParam` 从关联在 HTTP 头部的 cookie 里提取信息。

`@FormParam` 稍有不同，它所请求 MIME 媒体类型为 `application/x-www-form-urlencoded`，并且符合指定的 HTML 表单的编码。此参数提取对于 HTML 表单请求是非常有用的，例如，从发布的表单数据中提取名称是 `name` 的参数信息：

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("name") String name) {
    ...
}
```





如果需要查询路径参数，则从 Map 参数名称获取值，做法如下：

```
@GET
public String get(@Context UriInfo ui) {
    MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
    MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}
```

header 和 cookie 的参数用法如下：

```
@GET
public String get(@Context HttpHeaders hh) {
    MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
    Map<String, Cookie> pathParams = hh.getCookies();
}
```

@Context 一般用于获得一个 Java 类型的关联请求或响应的上下文。

form 表单参数（不像其他消息的一部分）是实体，做法如下：

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(MultivaluedMap<String, String> formParams) {
    ...
}
```

也就是说，不需要 @Context 注解。

另一种注入是 @BeanParam，允许在一个 bean 中注入上面所描述的参数。一个 @BeanParam 注解的 bean 中所有的字段和参数注解（像 @PathParam）将由相应的请求值来进行初始化（如果这些字段在资源类中）。@BeanParam 用于注入这种 bean 到资源或资源的方法中。@BeanParam 就是用这样的方式来聚集更多的请求参数到一个单一的 bean 中的。

下面是 @BeanParam 的用法示例：

```
public class MyBeanParam {
    @PathParam("p")
    private String pathParam;

    @MatrixParam("m")
    @Encoded
    @DefaultValue("default")
    private String matrixParam;
```





```
@HeaderParam("header")
private String headerParam;

private String queryParams;

public MyBeanParam(@QueryParam("q") String queryParams) {
    this.queryParam = queryParams;
}

public String getPathParam() {
    return pathParam;
}
...
}
```

将 MyBeanParam 以参数形式注入：

```
@POST
public void post(@BeanParam MyBeanParam beanParam, String entity) {
    final String pathParam = beanParam.getPathParam(); // 包含注入的路径参数 "p"
    ...
}
```

该例子展示了 @PathParam、@QueryParam、@MatrixParam 和 @HeaderParam 集中在一个 bean 里面的情况。@DefaultValue 用来定义矩阵参数的默认值。同时 @Encoded 注解都有同样的行为，用来在资源的方法上直接注入。将 bean 参数注入注解为 @Singleton 的资源类字段是不允许的（注入方法的参数必须替换）。

@BeanParam 可以包含所有的注入参数——@PathParam、@QueryParam、@MatrixParam、@HeaderParam、@CookieParam、@FormParam。

多个 bean 可以被注入一个资源或方法的参数，即使它们注入的是相同的请求值。例如，以下示例是有可能的：

```
@POST
public void post(@BeanParam MyBeanParam beanParam, @BeanParam AnotherBean
anotherBean, @PathParam("p") pathParam, String entity) {
    // beanParam.getPathParam() 等于 pathParam
    ...
}
```

3. 子资源

@Path 可以用在类上，这样的类称为根资源类；也可以用在根资源类的方法上，这使得许





多资源的方法被组合在一起，能够被重用。

第一种用法，`@Path` 用在资源的方法上，这类方法称为子资源方法（sub-resource method）。下面是显示一个资源类 `jmaki` 后端签名方法的示例：

```
@Singleton
@Path("/printers")
public class PrintersResource {

    @GET
    @Produces({"application/json", "application/xml"})
    public WebResourceList getMyResources() { ... }

    @GET @Path("/list")
    @Produces({"application/json", "application/xml"})
    public WebResourceList getListOfPrinters() { ... }

    @GET @Path("/jMakiTable")
    @Produces("application/json")
    public PrinterTableModel getTable() { ... }

    @GET @Path("/jMakiTree")
    @Produces("application/json")
    public TreeModel getTree() { ... }

    @GET @Path("/ids/{printerid}")
    @Produces({"application/json", "application/xml"})
    public Printer getPrinter(@PathParam("printerid") String printerId)
    { ... }

    @PUT @Path("/ids/{printerid}")
    @Consumes({"application/json", "application/xml"})
    public void putPrinter(@PathParam("printerid") String printerId,
Printer printer) { ... }

    @DELETE @Path("/ids/{printerid}")
    public void deletePrinter(@PathParam("printerid") String printerId)
    { ... }
}
```





如果请求 URL 的路径是 “printers”，那么在资源的方法中没有 `@Path` 注解的将被选择。如果请求 URL 的路径是 “printers/list”，则首先在根资源类中进行匹配，然后在子资源中匹配的方法 “list” 将被选择，在这种情况下，子资源方法是 `getListOfPrinters`。因此，在这个例子中的 URL 路径将会分层进行匹配。

第二种用法，`@Path` 可能用在那些没有用资源指示器（像 `@GET` 或 `@POST`）注解的方法上。这种方法被称为子资源定位器（sub-resource locator）。下面的示例显示了一个根资源类和乐观并发（optimistic-concurrency）的资源类的方法签名：

```
@Path("/item")
public class ItemResource {
    @Context UriInfo uriInfo;

    @Path("content")
    public ItemContentResource getItemContentResource() {
        return new ItemContentResource();
    }

    @GET
    @Produces("application/xml")
    public Item get() { ... }
}

public class ItemContentResource {

    @GET
    public Response get() { ... }

    @PUT
    @Path("{version}")
    public void put(@PathParam("version") int version,
        @Context HttpHeaders headers,
        byte[] in) {
        ...
    }
}
```

根资源类 `ItemResource` 包含子资源定位器 `getItemContentResource`，用于返回一个新的资源





类。如果请求 URL 的路径是“item/content”，则首先在根资源中进行匹配，而后子资源定位器将会匹配和调用，它将返回 `ItemContentResource` 资源类的一个实例。子资源定位器使得资源类能够被重用。方法上可以有空路径的 `@Path` 注解，如 `@Path("/")` 或 `@Path("")`，这意味着子资源定位器匹配了一个封闭的资源路径匹配（无子资源的路径）。

```
@Path("/item")
public class ItemResource {

    @Path("/")
    public ItemContentResource getItemContentResource() {
        return new ItemContentResource();
    }
}
```

在上面的例子中，子资源定位器 `getItemContentResource` 将匹配的请求路径是“/item/locator”或“/item”。

此外，由于资源类中由子资源定位器在运行时返回处理结果，从而支持多态性。子资源定位器返回什么样的子类型，取决于请求（例如，一次资源定位器可以返回什么样的子类型取决于不同的认证请求）。例如，下面的子资源定位器是有效的：

```
@Path("/item")
public class ItemResource {

    @Path("/")
    public Object getItemContentResource() {
        return new AnyResource();
    }
}
```

注意，运行时将没有生命周期管理，也不会子资源定位器所返回的实例中执行字段注入，这是因为运行时不知道实例的生命周期是什么。如果必须要将运行时管理子资源作为标准的资源，则类应按以下示例返回：

```
import javax.inject.Singleton;

@Path("/item")
public class ItemResource {
    @Path("content")
    public Class<ItemContentSingletonResource> getItemContentResource() {
        return ItemContentSingletonResource.class;
    }
}
```





```
    }  
}  
  
@Singleton  
public class ItemContentSingletonResource {  
    // 这个类将受到单个生命周期的管理  
}
```

默认情况下，JAX-RS 资源在每个请求范围内受到管理，这意味着将为每个请求创建新的资源。在这个例子中，`javax.inject.Singleton` 注解的意思是，资源将是单例模式，不受请求范围的管理。子资源定位器返回一个类，这意味着运行时将托管资源的实例及其生命周期。相反，如果方法返回的是实例，那么注释将没有效果，返回的实例将被使用。

子资源定位器也可以返回一个 `programmable resource model`（可编程的资源模型）。更多关于可编程资源模型的构建，请读者自行参阅官方文档 `resource builder` 这一部分的内容（网址见 <https://jersey.java.net/documentation/latest/resource-builder.html>）。下面的示例显示了子资源定位器返回非常简单的资源。

```
import org.glassfish.jersey.server.model.Resource;  
  
@Path("/item")  
public class ItemResource {  
  
    @Path("content")  
    public Resource getItemContentResource() {  
        return Resource.from(ItemContentSingletonResource.class);  
    }  
}
```

上面的代码与之前的例子有同样的效果。`Resource` 是一种来自 `ItemContentSingletonResource` 构造的简单资源。只要是有效的资源，都可以返回更复杂的编程化资源。

8.1.4 例子：用 SSE 构建实时 Web 应用

在标准的 HTTP 请求—响应下，客户端打开一个连接，发送一个 HTTP 请求（例如，HTTP GET 请求）到服务端，然后接收 HTTP 返回的响应，一旦这个响应完全被发送或接收，服务端就关闭连接。通常这种请求总是由一个客户端发起的。

相反，Server-Sent Events（SSE）是一种机制，一旦由客户端建立客户端到服务器之间的连





接，就能让服务端异步地将数据从服务端推送到客户端。当连接由客户端建立完成，服务端就提供数据，并决定当新数据“块”可用时将其发送到客户端。当一个新的数据事件在服务端产生时，这个事件被服务端发送到客户端，因此被称为 Server-Sent Events（服务器推送事件）。

SSE 通常重用一個连接来处理多个消息（称为事件）。SSE 还定义了一个专门的媒体类型 `text/event-stream`，描述一个从服务端发送到客户端的简单格式。SSE 还提供在大多数现代浏览器里的标准 JavaScript 客户端 API 实现。关于 SSE 的更多信息，请参见 SSE API 规范（<http://www.w3.org/TR/2009/WD-eventsource-20091029/>）。图 8-1 展示的是目前各个主流浏览器对 SSE 的支持情况（虚线框中的表示支持）。

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
			31						
			42						
			45						
			47						
			49						
6			50					4.3	
8		6	51					4.4	
9		8	52					4.4.4	
10		48				9.3			
11	14	49	53	10	39	10	all	52	51
		50	54	TP	40				
		51	55		41				
		52	56						

图 8-1 主流浏览器对 SSE 的支持情况

SSE 适合应用于服务端单向推送信息到客户端的场景。Jersey 的 SSE 大致可以分为发布—订阅模式和广播模式。

为了使用 Jersey 的 SSE 功能，需要在应用中添加如下依赖：

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-sse</artifactId>
</dependency>
```

1. 发布—订阅模式

服务端代码：

```
@Path("see-events")
public class SseResource {

    private EventOutput eventOutput = new EventOutput();
    private OutboundEvent.Builder eventBuilder;
```



```
private OutboundEvent event ;

/**
 * 提供 SSE 事件输出通道的资源方法
 * @return eventOutput
 */
@GET
@Produces(SseFeature.SERVER_SENT_EVENTS)
public EventOutput getServerSentEvents() {

    // 不断循环执行
    while (true) {
        SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        // 设置日期格式
        String now = df.format(new Date()); // 获取当前系统时间
        String message = "Server Time:" + now;
        System.out.println( message );

        eventBuilder = new OutboundEvent.Builder();
        eventBuilder.id(now);
        eventBuilder.name("message");
        eventBuilder.data(String.class,
            message ); // 推送服务器时间的信息给客户端
        event = eventBuilder.build();
        try {
            eventOutput.write(event);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                eventOutput.close();
                return eventOutput;
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

    }
}

```

上面的代码定义了部署在 URI “see-events” 上的资源。这个资源有一个 @GET 资源方法作为一个实体 EventOutput（Jersey ChunkedOutput API 的扩展用于输出分块消息的处理）。

客户端代码：

```

// 判断浏览器是否支持 EventSource
if (typeof (EventSource) !== "undefined") {
    var source = new EventSource("webapi/see-events");

    // 当通往服务器的连接被打开时
    source.onopen = function(event) {
        console.log("连接开启！");
    };

    // 收到消息，只监听命名是 message 的事件
    source.onmessage = function(event) {
        console.log(event.data);
        var data = event.data;
        var lastEventId = event.lastEventId;
        document.getElementById("x").innerHTML += "\n" + 'lastEventId:' +
lastEventId+';data:'+data;
    };

    // 可以是任意命名的事件名称
    /*
    source.addEventListener('message', function(event) {
        console.log(event.data);
        var data = event.data;
        var lastEventId = event.lastEventId;
        document.getElementById("x").innerHTML += "\n" + 'lastEventId:' +
lastEventId+';data:'+data;
    });
    */
}

```

```

// 错误发生
source.onerror = function(event) {
    console.log("连接错误!");
};

} else {
    document.getElementById("result").innerHTML = "Sorry, your browser does
not support server-sent events..."
}

```

首先要判断浏览器是否支持 EventSource，而后 EventSource 对象分别监听 onopen、onmessage、onerror 事件。其中，source.onmessage = function(event) {} 和 source.addEventListener('message', function(event) {}) 是一样的，区别是后者可以支持监听不同名称的事件，而 onmessage 属性只支持一个事件处理方法。

运行项目：

```
mvn jetty:run
```

在浏览器中访问 <http://localhost:8080>，如图 8-2 所示。

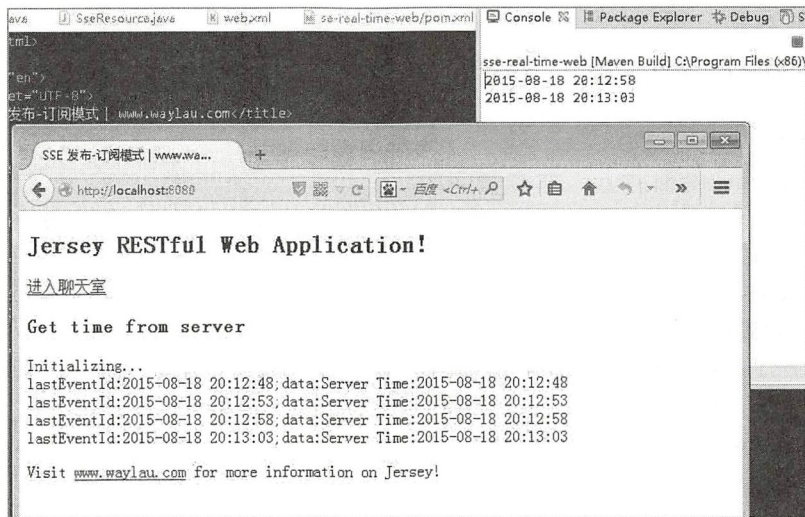


图 8-2 访问 <http://localhost:8080>

2. 广播模式

服务端代码：

```
@Singleton
```

```
@Path("sse-chat")
public class SseChatResource {

    private SseBroadcaster broadcaster = new SseBroadcaster();

    /**
     * 提供 SSE 事件输出通道的资源方法
     * @return eventOutput
     */
    @GET
    @Produces(SseFeature.SERVER_SENT_EVENTS)
    public EventOutput listenToBroadcast() {
        EventOutput eventOutput = new EventOutput();
        this.broadcaster.add(eventOutput);
        return eventOutput;
    }

    /**
     * 提供写入 SSE 事件通道的资源方法
     * @param message
     * @param name
     */
    @POST
    @Produces(MediaType.TEXT_PLAIN)
    public void broadcastMessage(@DefaultValue("waylau.com") @QueryParam
("message") String message,
        @DefaultValue("waylau") @QueryParam("name") String name) {
        // 设置日期格式
        SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String now = df.format(new Date()); // 获取当前系统时间
        message = now + ":" + name + ":" + message; // 发送的消息带上当前的时间

        OutboundEvent.Builder eventBuilder = new OutboundEvent.Builder();
        OutboundEvent event = eventBuilder.name("message")
            .mediaType(MediaType.TEXT_PLAIN_TYPE)
            .data(String.class, message)
            .build();
    }
}
```



```

        // 发送广播
        broadcaster.broadcast(event);
    }
}

```

其中, `SseChatResource` 资源类用 `@Singleton` 注解, 告诉 Jersey 运行时资源类只有一个实例, 用于传入 `/sse-chat` 路径的所有请求。应用程序引用私有的 `broadcaster` 字段, 这样所有请求可以使用相同的实例。客户端想监听 SSE 事件, 先发送 GET 请求到 `sse-chat` 的 `listenToBroadcast()` 资源方法进行处理。方法创建一个新的 `EventOutput` 用于展示请求的客户端的连接, 并通过 `add(EventOutput)` 注册 `eventOutput` 实例到单例 `broadcaster`。方法返回 `eventOutput` 实例导致 Jersey 请求的客户端事件与 `eventOutput` 实例绑定, 向客户机发送响应 HTTP 头。客户端连接保持开放, 客户端准备接收新的 SSE 事件。所有的事件通过 `broadcaster` 写入 `eventOutput` 实例。这样开发人员可以方便地发送新的事件到所有订阅的客户端。

当客户端想要广播新消息给所有已经监听 SSE 连接的客户端时, 它先发送一个 POST 请求, 将消息内容发到 `SseChatResource` 资源。`SseChatResource` 资源调用方法 `broadcastMessage`, 消息内容作为输入参数。一个新的 SSE 出站事件建立在标准方法上并传递给 `broadcaster`。`broadcaster` 内部在所有注册了的 `EventOutput` 上调用 `write(OutboundEvent)`。该方法只返回一个标准文本响应给客户端, 来通知客户端已经成功广播了消息。正如我们所看到的, `broadcastMessage` 资源方法只是一个简单的 JAX-RS 资源的方法。

Jersey `SseBroadcaster` 完成该用例不是强制性的。每个 `EventOutput` 可以只存储在收集器里, 在 `broadcastMessage` 方法里面迭代。然而, `SseBroadcaster` 内部会识别和处理客户端断开连接。当客户端关闭连接时, `broadcaster` 可检测并删除过期的在内部收集器里面注册 `EventOutput` 的连接, 以及释放所有服务器端关联陈旧连接的资源。此外, `SseBroadcaster` 的实现是线程安全的, 这样客户端可以在任何时间连接和断开, `SseBroadcaster` 总是广播消息给最近收集的注册的和活跃的客户。

客户端代码:

```

// 判断浏览器是否支持 EventSource
if (typeof (EventSource) !== "undefined") {
    var source = new EventSource("webapi/sse-chat");

    // 当通往服务器的连接被打开时
    source.onopen = function(event) {
        var ta = document.getElementById('response_text');
        ta.value = '连接开启!';
    };
}

```

```
// 收到消息，只监听命名是 message 的事件
source.onmessage = function(event) {
    var ta = document.getElementById('response_text');
    ta.value = ta.value + '\n' + event.data;
};

// 可以是任意命名的事件名称
/*
source.addEventListener('message', function(event) {
    var ta = document.getElementById('response_text');
    ta.value = ta.value + '\n' + event.data;
});
*/

// 错误发生
source.onerror = function(event) {
    var ta = document.getElementById('response_text');
    ta.value = ta.value + '\n' + "连接出错! ";
};
} else {
    alert("Sorry, your browser does not support server-sent events");
}

function send(message) {
    var xmlhttp;
    var name = document.getElementById('name_id').value;

    if (window.XMLHttpRequest)
    { // 适用于 IE7+、Firefox、Chrome、Opera、Safari
        xmlhttp=new XMLHttpRequest();
    }
    else
    { // 适用于 IE6、IE5
        xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
    }

    xmlhttp.open("POST", "webapi/sse-chat?message=" + message + '&name=' +
name ,true);
    xmlhttp.send();
}
```

EventSource 的用法与发布—订阅模式类似。而 `send(message)` 方法将消息以 POST 请求发送给服务器端，而后将该消息进行广播，从而实现聊天室的效果。

最终效果如图 8-3 所示。



图 8-3 最终效果

上面例子的代码可以在 <https://github.com/waylau/distributed-systems-technologies-and-cases-analysis> 的 `sse-real-time-web` 程序中找到。

8.2 Spring Boot

Spring Boot 旨在简化创建产品级的 Spring 应用和服务，可通过它来选择不同的 Spring 平台。Spring Boot 为 Spring 平台及第三方库提供开箱即用的设置，这样我们可以有条不紊地开始开发工作。多数 Spring Boot 应用只需要很少的 Spring 配置。Spring Boot 可创建独立的 Java 应用和 Web 应用，可以使用 `java -jar` 启动它或采用传统的 WAR 部署方式。同时提供“spring scripts”命令行工具。

Spring Boot 也是构建微服务的框架，在 Spring Boot 中实现一个基于 HTTP 的 RESTful 微服务，只需简单地加入 actuator 与 Web 启动模块就足够了。

截至目前，Spring Boot 的最新版本为 2.0.2。Spring Boot 的官方网站为 <http://projects.spring.io/spring-boot/>。本节只对 Spring Boot 做简单的介绍，如果读者想了解有关 Spring Boot 的企业级开发方式，可以参阅笔者所著的《Spring Boot 企业级应用开发实战》。

8.2.1 Spring Boot 简介

多年以来，Spring 平台饱受非议的一点就是大量的 XML 配置及复杂的依赖管理。随着 Spring 3.0 的发布，Spring IO 团队逐渐开始摆脱 XML 配置文件，并且在开发过程中大量使用“约定大于配置（convention over configuration）”的思想来摆脱 Spring 框架中各类繁杂的配置（即 Java Config）。

Spring Boot 正是在这样的背景下被抽象出来的开发框架，它本身并不提供 Spring 框架的核心特性及扩展功能，只是用于快速、敏捷地开发新一代基于 Spring 框架的应用程序。也就是说，它并不是用来替代 Spring 的解决方案，而是和 Spring 框架紧密结合用于提升 Spring 开发者体验的工具。同时它集成了大量常用的第三方库配置，在 Spring Boot 应用中，这些第三方库几乎可以零配置地开箱即用，大部分 Spring Boot 应用只需要非常少量的配置代码，这样开发者能够更加专注于业务逻辑。

在追求开发体验的提升方面，Spring Boot，甚至可以说整个 Spring 生态系统都使用了 Groovy 编程语言。Spring Boot 所提供的众多便捷功能，都借助于 Groovy 强大的 MetaObject 协议、可插拔的 AST 转换过程及内置了解决方案引擎所实现的依赖。在其核心的编译模型中，Spring Boot 使用 Groovy 来构建工程文件，所以它可以使用通用的导入和样板方法（如类的 main 方法）对类所生成的字节码进行装饰（decorate）。这样使用 Spring Boot 编写的应用就能保持非常简洁，却依然可以提供众多的功能。

Spring Boot 项目主要的目的是：

- 为 Spring 的开发提供更快、更广泛的快速上手帮助；
- 使用默认方式实现快速开发；
- 提供大多数项目所需的非功能特性，诸如嵌入式服务器、安全、心跳检查、外部配置等；
- 绝对没有代码生成（code generation），也不需要 XML 配置。

Spring Boot 内嵌容器以支持开箱即用，如表 8-1 所示。

表 8-1 Spring Boot 内嵌的容器

名 称	Servlet 版本	Java 版本
Tomcat 8	3.1	Java 7+
Tomcat 7	3.0	Java 6+
Jetty 9.3	3.1	Java 8+
Jetty 9.2	3.1	Java 7+
Jetty 8	3.0	Java 6+
Undertow 1.3	3.1	Java 7+

也可以将 Spring Boot 应用部署到任何兼容 Servlet 3.0+ 的容器。

8.2.2 Spring Boot 的安装

从根本上来讲，Spring Boot 就是一些库的集合，它能够被任意项目的构建系统所使用。简便起见，该框架也提供了命令行界面，它可以用来运行和测试 Spring Boot 应用。

不管怎样，都需要安装 Java SDK v1.6 或更高版本。尽管 Spring Boot 兼容 Java 1.6，不过最好使用 Java 的最新版本。

1. 为 Java 开发者准备的安装方式

对于 Java 开发者来说，使用 Spring Boot 就跟使用其他 Java 库一样，只需要在 classpath 下引入适当的 spring-boot-*.jar 文件即可。Spring Boot 不需要集成任何特殊的工具，所以可以使用任意 IDE 或文本编辑器；同时，Spring Boot 应用也没有什么特殊之处，我们可以像对待其他 Java 程序那样运行、调试它。

尽管可以直接复制 Spring Boot jar 文件，但还是建议使用支持依赖管理的构建工具，比如 Maven 或 Gradle。

通过 Maven 方式安装

Spring Boot 兼容 Apache Maven 3.2 或更高版本。如果本地没有安装 Maven，则可以参考 maven.apache.org 上的指南。

注：在很多操作系统上，可以通过包管理器来安装 Maven。OSX Homebrew 用户可以使用 `brew install maven`，Ubuntu 用户可以运行 `sudo apt-get install maven`。

Spring Boot 依赖使用的 groupId 为 org.springframework.boot。通常 Maven POM 文件会继承 spring-boot-starter-parent 工程，并声明一个或多个“Starter”依赖。此外，Spring Boot 提供了一个可选的 Maven 插件，用于创建可执行的 qjar。

下面是一个典型的 pom.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>myproject</artifactId>
```

```
<version>0.0.1-SNAPSHOT</version>

<!-- 默认从 Spring Boot 继承 -->
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.2.RELEASE</version>
</parent>

<!-- 添加一个 Web 应用的典型依赖 -->
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>

<!-- 打包为一个可执行的 jar -->
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

</project>
```

注：spring-boot-starter-parent 是使用 Spring Boot 的一种不错的方式，但它并不总是最合适的。有时可能需要继承一个不同的父 POM，或者只是不喜欢默认配置，可以使用 import 作用域这种替代方案。

通过 Gradle 方式安装

Spring Boot 兼容 Gradle 的最新版本。如果本地没有安装 Gradle，则可以参考 www.gradle.org 上的指南。另外笔者也对官方指南进行了翻译，可作为参考，网址为 <https://github.com/waylau/Gradle-2-User-Guide>。

Spring Boot 的依赖可通过 groupId org.springframework.boot 来声明。通常项目将声明一个或

多个“Starter”依赖。Spring Boot 提供了一个很有用的 Gradle 插件，可以用来简化依赖声明，创建可执行的 jar。

注：当需要构建项目时，Gradle Wrapper 提供一种给力的获取 Gradle 的方式。它是一小段脚本和库，跟代码一块提交，用于启动构建进程，具体可参考 Gradle Wrapper 文档（网址为 www.gradle.org/docs/current/userguide/gradle_wrapper.html）。

下面是一个典型的 build.gradle 文件：

```
buildscript {
    repositories {
        jcenter()
        maven { url "http://repo.spring.io/snapshot" }
        maven { url "http://repo.spring.io/milestone" }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:
2.0.2.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'spring-boot'

jar {
    baseName = 'myproject'
    version = '0.0.1-SNAPSHOT'
}

repositories {
    jcenter()
    maven { url "http://repo.spring.io/snapshot" }
    maven { url "http://repo.spring.io/milestone" }
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile("org.springframework.boot:spring-boot-starter-test")
}
```

2. Spring Boot CLI 的安装

Spring Boot CLI 是一个命令行工具，可用于快速搭建基于 Spring 的原型。它支持运行 Groovy 脚本，这也就意味着可以使用类似 Java 的语法，但不用写很多模板代码。

Spring Boot 不一定非要配合 CLI 使用，但它绝对是 Spring 应用取得进展的最快方式。

手动安装

Spring CLI 分发包可以从 Spring 软件仓库下载：

- spring-boot-cli-2.0.2.RELEASE-bin.zip;
- spring-boot-cli-2.0.2.RELEASE-bin.tar.gz。

不稳定的 snapshot 分发包也可以从 <http://repo.spring.io/snapshot/org/springframework/boot/spring-boot-cli/> 获取。

下载完成后，解压分发包，根据存档里的 INSTALL.txt 操作指南进行安装。总的来说，在 .zip 文件的 bin/ 目录下会有一个 Spring 脚本（Windows 下是 spring.bat），或者使用 java -jar 运行 lib/ 目录下的 jar 文件（该脚本会确保 classpath 被正确设置）。

使用 SDKMAN! 进行安装

SDKMAN! 是一个软件开发工具管理器，可以对各种各样的二进制 SDK 包进行版本管理，包括 Groovy 和 Spring Boot CLI。可以从 <http://sdkman.io> 下载 SDKMAN!，并使用以下命令安装 Spring Boot：

```
$ sdk install springboot
$ spring --version
Spring Boot v2.0.2.RELEASE
```

如果正在为 CLI 开发新的特性，并想轻松获取刚构建的版本，则可以使用以下命令：

```
$ sdk install springboot dev /path/to/spring-boot/spring-boot-cli/target/
spring-boot-cli-2.0.2.RELEASE-bin/spring-2.0.2.RELEASE/
$ sdk default springboot dev
$ spring --version
Spring CLI v2.0.2.RELEASE
```

这将会安装一个名叫 dev 的本地 Spring 实例，它指向目标构建位置，所以每次重新构建 Spring Boot 时，Spring 都会更新为最新的版本。

可以通过以下命令来验证：

```
$ sdk ls springboot
```

=====


```
Available Springboot Versions
```

```
=====
> + dev
* 2.0.2.RELEASE
```

```
=====
+ - local version
* - installed
> - currently in use
=====
```

使用 OSX Homebrew 进行安装

如果你的环境是 Mac OS，使用的是 Homebrew，那么安装 Spring Boot CLI 只需以下操作：

```
$ brew tap pivotal/tap
$ brew install springboot
```

Homebrew 将把 Spring 安装到/usr/local/bin 下。

注：如果该方案不可用，则可能是因为 brew 版本太老了。只需执行 brew update 并重试即可。

使用 MacPorts 进行安装

如果环境是 Mac OS，使用的是 MacPorts，那么安装 Spring Boot CLI 只需以下操作：

```
$ sudo port install spring-boot-cli
```

命令行实现

Spring Boot CLI 启动脚本为 BASH 和 zsh shells 提供完整的命令行实现，可以放在任何 shell 的 source 脚本（名称也是 spring）中，或者放到用户/系统范围内 bash 初始化脚本中。在 Debian 系统中，系统级的脚本位于 /shell-completion/bash 下，当新的 shell 启动时，该目录下的所有脚本都会被执行。如果想手动运行脚本，假如已经安装了 SDKMAN!，则可以使用以下命令：

```
$ . ~/.sdkman/candidates/springboot/current/shell-completion/bash/spring
$ spring <HIT TAB HERE>
grab help jar run test version
```

注：如果使用 Homebrew 或 MacPorts 安装 Spring Boot CLI，则命令行实现脚本会自动注册到 shell 中。

Spring Boot CLI 示例

下面是一个简单的 Web 应用，可以用它测试 Spring Boot CLI 的安装是否成功。创建一个名

为 app.groovy 的文件：

```
@RestController
class ThisWillActuallyRun {

    @RequestMapping("/")
    String home() {
        "Hello World!"
    }

}
```

然后只需在 shell 中运行以下命令即可：

```
$ spring run app.groovy
```

注：首次运行该应用将会花费一些时间，因为需要下载依赖，后续运行会快很多。

使用浏览器打开 localhost:8080，然后就可以看到如下输出：

```
Hello World!
```

8.2.3 Spring Boot 的使用

本节将使用 Java 开发一个简单的“Hello World”Web 应用，以此来展示 Spring Boot 的一些关键特性。项目采用 Maven 进行构建。

在开始前，需要打开终端检查安装的 Java 和 Maven 版本是否可用：

```
$ java -version
java version "1.8.0_112"
Java(TM) SE Runtime Environment (build 1.8.0_112-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.112-b15, mixed mode)
$ mvn -v
Apache Maven 3.5.2 (138edd61fd100ec658bfa2d307c43b76940a5d7d;
2017-10-18T15:58:13+08:00)
Maven home: D:\Program Files\apache-maven-3.5.2\bin\..
Java version: 1.8.0_162, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.8.0_162\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

1. 创建 POM

先创建一个 Maven pom.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>myproject</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.2.RELEASE</version>
    </parent>

    <!-- 其他的加在下面... -->

</project>
```

这样一个可工作的项目就构建完成了，可以通过运行 `mvn package` 来测试它。暂时出现“jar will be empty - no content was marked for inclusion (jar 将是空的——没有包含任何内容)”警告。

此刻，可以将该项目导入 IDE（大多数现代的 Java IDE 都包含对 Maven 的内建支持）。

2. 添加 classpath 依赖

Spring Boot 提供很多“Starter”，用来简化添加 jar 到 classpath 的操作。在示例程序中已经在 POM 的 `parent` 节点使用了 `spring-boot-starter-parent`，它是一个特殊的“Starter”，提供有用的 Maven 默认设置。同时，它也提供一个 `dependency-management` 节点，这样对于“blessed”的依赖就可以省略 `version` 标记了。

其他“Starter”只简单提供开发特定类型应用所需的依赖。由于正在开发 Web 应用，所以添加 `spring-boot-starter-web` 依赖。但在此之前，先看一下目前的依赖：

```
$ mvn dependency:tree
```

```
[INFO] com.example:myproject:jar:0.0.1-SNAPSHOT
```

`mvn dependency:tree` 命令可以将项目依赖以树形方式展现出来，可以看到 `spring-boot-starter-parent` 本身并没有提供依赖。编辑 `pom.xml`，并在 `parent` 节点下添加 `spring-boot-starter-web` 依赖：

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

如果再次运行 `mvn dependency:tree`，将看到现在多了一些其他依赖，包括 Tomcat Web 服务器和 Spring Boot 自身。

3. 编写代码

为了完成应用程序，我们需要创建一个单独的 Java 文件。Maven 默认会编译 `src/main/java` 下的源码，所以需要创建那样的文件结构，并添加一个名为 `src/main/java/Example.java` 的文件：

```
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Example {

    @RequestMapping("/")
    String home() {
        return "Hello World!";
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(Example.class, args);
    }
}
```

@RestController 和 @RequestMapping 注解

`Example` 类上使用的第一个注解是 `@RestController`，被称为 `stereotype` 注解。它为阅读代码的人提供暗示，这是一个支持 REST 的控制器。在本示例中，我们的类是一个 Web 的 `@Controller`，

所以当 Web 请求进来时，Spring 会考虑是否使用它来进行处理。

@RequestMapping 注解提供路由信息，它告诉 Spring 任何来自 “/” 路径的 HTTP 请求都应该被映射到 home 方法上。@RestController 注解告诉 Spring 以字符串的形式渲染结果，并直接返回给调用者。

注：@RestController 和 @RequestMapping 是 Spring MVC 中的注解（它们不是 Spring Boot 的特定部分），具体可以参考 Spring 官方文档（见 <http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/>），或者参考笔者所写的中文翻译版《Spring Framework 4.x 参考文档》（见 <https://github.com/waylau/spring-framework-4-reference>）。

@EnableAutoConfiguration 注解

第二个类级别的注解是@EnableAutoConfiguration，这个注解告诉 Spring Boot 根据添加的 jar 依赖猜测用户想要如何配置 Spring。由于 spring-boot-starter-web 添加了 Tomcat 和 Spring MVC，所以 auto-configuration 将假定用户正在开发一个 Web 应用，并对 Spring 进行相应的设置。

auto-configuration 设计成可以跟 “Starter” 一起搭配使用，但这两个概念没有直接的联系。可以自由地挑选 Starter 以外的 jar 依赖，Spring Boot 仍会尽最大努力去自动配置应用。

main 方法

应用程序的最后部分是 main 方法，这是一个标准的方法，它遵循 Java 对于一个应用程序入口的约定。main 方法通过调用 run，将业务委托给 Spring Boot 的 SpringApplication 类。SpringApplication 将引导应用启动 Spring，相应地启动被自动配置的 Tomcat Web 服务器。我们需要将 Example.class 作为参数传递给 run 方法，以此告诉 SpringApplication 谁是主要的 Spring 组件，并传递 args 数组以暴露所有的命令行参数。

4. 运行示例

由于使用 spring-boot-starter-parent 的 POM，这样就可以通过 run 目标来启动程序。在项目的根目录下输入 mvn spring-boot:run 来启动应用：

```
$ mvn spring-boot:run
```

```

      ____  _
     / __ \| | | |
    / /_ \| | | |
   / ___ \| | | |
  /_/   \_\_|_|_|

:: Spring Boot :: (v2.0.2.RELEASE)

```

```

.....
..... (log output here)
.....
..... Started Example in 2.222 seconds (JVM running for 6.514)

```

使用浏览器打开 `localhost:8080`，应该可以看到如下输出：

```
Hello World!
```

使用 `Ctrl+C` 组合键关闭应用程序。

5. 创建一个可执行 jar

可执行 jar 有时被称为“fat jar”（胖 jar），是包含编译后的类及代码运行所需依赖 jar 的存档。

Java 没有提供任何标准方式用来加载内嵌的 jar 文件（即 jar 文件中还包含 jar 文件），这对分发自包含应用来说是个问题。为了解决该问题，很多开发者采用“共享的”jar。共享的 jar 只是简单地将所有 jar 的类打包进一个单独的存档。这种方式存在的问题是，很难区分应用程序中使用了哪些库。在多个 jar 中如果存在相同的文件名（但内容不一样）也会出现问题。Spring Boot 采取一种不同的方式，允许用户真正地直接内嵌 jar。如果读者对可执行 jar 有兴趣，可以自行参阅在线文档（<http://docs.spring.io/spring-boot/docs/current/reference/html/executable-jar.html>）。

为了创建可执行的 jar，我们需要将 `spring-boot-maven-plugin` 添加到 `pom.xml` 中，在 `dependencies` 节点后面插入以下内容：

```

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

```

注：`spring-boot-starter-parent` POM 包含绑定到 `repackage` 目标的 `<executions>` 配置。如果不使用父 POM，则需要自己声明该配置，具体参考插件文档（<http://docs.spring.io/spring-boot/docs/2.0.2.RELEASE/maven-plugin/usage.html>）。

保存 `pom.xml`，并从命令行运行 `mvn package`：

```
$ mvn package
```

```
[INFO] Scanning for projects...
```

```
[INFO] -----
[INFO] Building myproject 0.0.1-SNAPSHOT
[INFO] -----
[INFO] .... ..
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ myproject ---
[INFO] Building jar: /Users/developer/example/spring-boot-example/target/
myproject-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.0.2.RELEASE:repackage (default) @
myproject ---
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

如果查看 `target` 目录，则应该可以看到 `myproject-0.0.1-SNAPSHOT.jar`，该文件大概有 10 MB。想查看内部结构，可以运行 `jar tvf`：

```
$ jar tvf target/myproject-0.0.1-SNAPSHOT.jar
```

在该目录下, 应该还能看到一个很小的名为 `myproject-0.0.1-SNAPSHOT.jar.original` 的文件, 这是在 Spring Boot 重新打包前 Maven 创建的原始 jar 文件。

可以使用 `java -jar` 命令运行该应用程序:

```
$ java -jar target/myproject-0.0.1-SNAPSHOT.jar
```

```
. _____ _ _ _ _  
/\ \ / ____' _ _ _ _ _ ( ) _ _ _ _ _ \ \ \ \ \  
( ( ) \__ | ' _ | ' _ | | ' _ \ / _ ' | \ \ \ \  
\ \ / ____ | |_) | | ! | | | | ( _ | ) ) ) )  
 ' |____| . __ | | | _ | | \__, | / / / /  
=====|_|=====|___/=/_/_/_/  
  
:: Spring Boot :: (v2.0.2.RELEASE)  
  
.....  
..... (log output here)  
.....  
..... Started Example in 2.536 seconds (JVM running for 2.864)
```

使用 Ctrl+C 组合键可以退出应用。

8.2.4 Spring Boot 的属性与配置

Spring Boot 允许通过外部配置在不同的环境中使用同一应用程序的代码，简单说就是可以通过配置文件来注入属性或修改默认的配置。

Spring Boot 提供一种优先级配置读取的机制。Spring Boot 使用一个非常特别的 `PropertySource` 次序来允许对值进行合理的覆盖，需要以下面的次序考虑属性：

- devtools 的全局设置属性（当 devtools 激活后，在用户 home 目录 `~/.spring-boot-devtools.properties` 下）；
- 测试中的 `@TestPropertySource` 注解；
- 测试中的 `@SpringBootTest#properties` 注解属性；
- 命令行参数；
- `SPRING_APPLICATION_JSON` 中的属性（内联 JSON 内嵌在环境变量或系统属性中）；
- `ServletConfig` 初始化参数；
- `ServletContext` 初始化参数；
- 来自 `java:comp/env` 的 JNDI 属性；
- Java 系统属性（`System.getProperties()`）；
- 操作系统环境变量；
- 只有在 `random.*` 里包含的属性才会产生一个 `RandomValuePropertySource`；
- 在打包的 jar 外的特定应用程序属性（`application-{profile}.properties` 和 YAML 变量）；
- 在打包的 jar 内的特定应用程序属性（`application-{profile}.properties` 和 YAML 变量）；
- 在打包的 jar 外的应用程序配置文件（`application.properties` 和 YAML 变量）；
- 在打包的 jar 内的应用程序配置文件（`application.properties` 和 YAML 变量）；
- 在 `@Configuration` 类上的 `@PropertySource` 注解；
- 默认属性（使用 `SpringApplication.setDefaultProperties` 指定）。

本节介绍 Spring Boot 的属性，以及各种配置方式。

1. 在构建时自动扩展属性

除了在项目构建配置中硬编码指定的一些属性，还可以使用现有构建配置自动扩展它们。这在 Maven 和 Gradle 中都能实现。

Maven

可以使用资源过滤从 Maven 项目中自动扩展属性。如果使用 spring-boot-starter-parent，则可以通过 “@…@” 占位符引用 Maven 的“项目属性”：

```
app.encoding=@project.build.sourceEncoding@
app.java.version=@java.version@
```

如果没有使用 starter parent，则可以在 pom.xml 中添加如下元素：

```
<resources>
  <resource>
    <directory>src/main/resources</directory>
    <filtering>true</filtering>
  </resource>
</resources>
```

并添加 plugin 插件：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-resources-plugin</artifactId>
  <version>2.7</version>
  <configuration>
    <delimiters>
      <delimiter>@</delimiter>
    </delimiters>
    <useDefaultDelimiters>false</useDefaultDelimiters>
  </configuration>
</plugin>
```

Gradle

可以通过配置 Java 插件的 processResources 任务来实现在 Gradle 项目中自动扩展属性：

```
processResources {
    expand(project.properties)
}
```

接着就可以引用 Gradle 项目的属性：

```
app.name=${name}
app.description=${description}
```

2. 外部化 SpringApplication 配置

SpringApplication 已经被属性化（主要是 setters），所以可以在创建应用时使用它的 Java API 来修改行为。或者可以使用 `spring.main.*` 中的属性来外部化这些配置。比如，在 `application.properties` 中可能会有以下内容：

```
spring.main.web-environment=false
spring.main.banner-mode=off
```

然后 Spring Boot 在启动时将不会显示 banner，并且该应用也不再是一个 Web 应用。

3. 改变应用程序外部配置文件的位置

为应用程序源添加 `@PropertySource` 注解是一种很好的添加和修改源顺序的方法。传递给 SpringApplication 静态方法的类和使用 `setSources()` 添加的类都会被检查，以查看它们是否有 `@PropertySources`。如果有，则这些属性会被尽可能早地添加到 Environment 里，以确保 ApplicationContext 生命周期的所有阶段都能使用。以这种方式添加的属性优先于任何使用默认位置添加的属性，但低于系统属性、环境变量或命令行参数。

也可以提供系统属性（或环境变量）来改变该行为。

- `spring.config.name`（`SPRING_CONFIG_NAME`）是根文件名，默认为 `application`。
- `spring.config.location`（`SPRING_CONFIG_LOCATION`）是要加载的文件（例如，一个 classpath 资源或一个 URL）。Spring Boot 为该文档设置一个单独的 Environment 属性，它可以被系统属性、环境变量或命令行参数覆盖。

不管 environment 被设置成什么，Spring Boot 都将加载上面讨论过的 `application.properties`。如果使用 YAML，则具有 “.yaml” 扩展的文件默认也会被添加到该列表中。

4. 使用 short 命令行参数

有些人喜欢使用 `--port=9000` 代替 `--server.port=9000` 来设置命令行配置属性。可以通过在 `application.properties` 中使用占位符来启用该功能，比如：

```
server.port=${port:8080}
```

注：如果继承自 `spring-boot-starter-parent` POM，则为了防止和 Spring 样式的占位符产生冲突，`maven-resources-plugins` 默认的过滤令牌（filter token）已经从 `${*}` 变为 `@`（即 `@maven.token@` 代替了 `${maven.token}`）。如果已经直接启用 Maven 对 `application.properties` 的过滤，则可能也想使用其他的分隔符替换默认的过滤令牌（详见 <http://maven.apache.org/plugins/maven-resources-plugin/resources-mojo.html#delimiters>）。

在这种特殊的情况下，端口绑定能够在一种 PaaS 环境下工作，比如 Heroku 和 Cloud Foundry，

因为在这两个平台中 PORT 环境变量是自动设置的，并且 Spring 能够绑定 Environment 属性的大写同义词。

5. 使用 YAML 配置外部属性

YAML 是 JSON 的一个超集，可以非常方便地将外部配置以层次结构形式存储起来。比如：

```
spring:
  application:
    name: cruncher
  datasource:
    driverClassName: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost/test
server:
  port: 9000
```

创建一个 application.yml 文件，将它放到 classpath 的根目录下，并添加 snakeyaml 依赖（Maven 坐标为 org.yaml:snakeyaml，如果使用 spring-boot-starter，就已经被包含了）。一个 YAML 文件会被解析为一个 Java 的 Map<String,Object>（和一个 JSON 对象类似），Spring Boot 会将该 map 做扁平化处理，这样它就只有 1 级深度，并且有 period-separated 的 key，与人们在 Java 中经常使用的 Properties 文件非常类似。

上面的 YAML 示例对应于下面的 application.properties 文件：

```
spring.application.name=cruncher
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/test
server.port=9000
```

6. 设置生效的 Spring profiles

Spring 的 Environment 有一个 API 可以设置生效的 profiles，但通常会设置一个系统属性（spring.profiles.active）或一个操作系统的环境变量（SPRING_PROFILES_ACTIVE）。比如，使用一个 -D 参数启动应用程序（记着把它放到 main 类或 jar 文件之前）：

```
$ java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar
```

在 Spring Boot 中，也可以在 application.properties 里设置生效的 profile，例如：

```
spring.profiles.active=production
```

通过这种方式设置的值会被系统属性或环境变量替换，但不会被 SpringApplicationBuilder.profiles()方法替换。因此，后面的 Java API 可用来在不改变默认设置的情况下增加 profiles。



7. 根据环境改变配置

一个 YAML 文件实际上是一系列以 “---” 线分割的文档，每个文档都被单独解析为一个扁平的 map。

如果一个 YAML 文档包含一个 `spring.profiles` 关键字，则 `profiles` 的值（以逗号分割的 `profiles` 列表）将被传入 Spring 的 `Environment.acceptsProfiles()` 方法，并且如果这些 `profiles` 的任何一个被激活，则对应的文档将被包含到最终的合并文件中（否则不会）。

示例：

```
server:
  port: 9000
---

spring:
  profiles: development
server:
  port: 9001
---

spring:
  profiles: production
server:
  port: 0
```

在这个示例中，默认的端口是 9000，如果 Spring profile 的 “development” 生效，则该端口是 9001；如果 “production” 生效，则它是 0。

YAML 文档以它们遇到的顺序合并（所以后面的值会覆盖前面的值）。

8. 发现外部属性的内置选项

Spring Boot 在运行时将来自 `application.properties`（或 `.yml`）的外部属性绑定进一个应用。不可能在一个地方存在详尽的支持所有属性的列表（技术上也是不可能的），因为 `classpath` 下的其他 `jar` 文件也能有配置文件。

每个运行中具有 Actuator 特性的应用都会有一个 `configprops` 端点，它能够展示所有边界和可通过 `@ConfigurationProperties` 绑定的属性。



8.3 Docker

Docker 是用 Go 语言编写的，自 2013 年推出以来，受到越来越多的开发者关注。无论云服务还是微服务（Microservices），越来越多的厂商都开始选择 Docker 作为基础设施自动化的工具。毫无疑问，Docker 是当前最热门的容器技术之一。

8.3.1 Docker 简介

Docker 提供一种方法来运行在容器中安全隔离的应用程序。应用程序与其所有的依赖和库将被一起打包，这样就确保应用程序总是可以使用它在构建映像中所期望的环境来运行，测试和部署比以往任何时候都更简单，因为整个构建过程将是完全可移植的，并且可以按照任何环境中的设计来运行。由于容器是轻量级的，运行的时候并没有虚拟机管理程序的额外负载，这样就可以运行许多应用程序，这些应用程序都依赖于单个内核上的不同库和环境，每个应用程序都不会互相干扰。将应用程序从虚拟机或物理机转移到容器实例中，可以获得更多的硬件资源。

典型的 Docker 平台工作流程：

- 将代码及其依赖项添加到 Docker 容器。
 - 编写一个指定执行环境并“pull”代码的 Dockerfile；
 - 如果应用程序依赖于外部应用程序（如 Redis 或 MySQL），则只需在 registry（例如，Docker Hub）中找到它们，在 Docker Compose 文件中引用它们并引用应用程序，以便它们同时运行，软件提供商还可以通过 Docker Store 分发付费软件；
 - 构建项目，然后在开发时通过 Docker Machine 在虚拟主机上运行容器；
- 如果需要，那么可以为解决方案配置网络和存储。
- 将构建版本上传到 registry（用户自己或其他云提供商的），以与团队协作。
- 如果需要在多个主机（VM 或物理机）上扩展解决方案，则应计划如何设置 Swarm 集群并扩展以满足需求。使用 Universal Control Plane，可以使用友好的界面来管理 Swarm 集群。
- 使用 Docker Cloud 部署到首选的云提供商（为了冗余，可以部署到多个云提供商），或者使用 Docker Datacenter 部署到用户的内部部署硬件。

8.3.2 Docker 的核心组成、架构及工作原理

本节介绍 Docker 的核心组成、架构及工作原理。同时简单介绍 Docker 所应用的底层技术。



1. Docker Engine

Docker Engine 是客户端—服务器模式的应用程序，有以下主要组件：

- 一个服务器，它是一种长时间运行的程序，称为守护进程。
- 一个 REST API，它指定程序可以用来与守护程序通信的接口，并可以指示它执行什么操作。
- 命令行界面（CLI）客户端。

Docker Engine 组件流如图 8-4 所示。

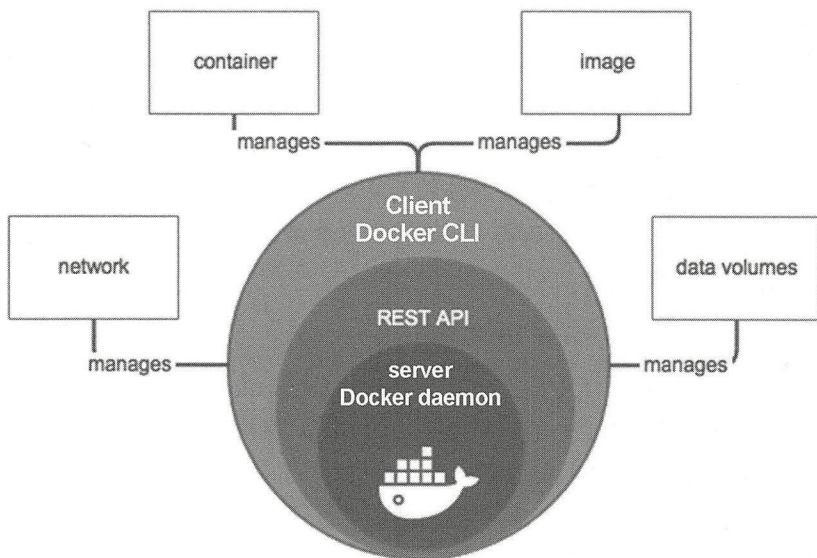


图 8-4 Docker Engine 组件流

CLI 使用 Docker REST API 通过脚本或 CLI 命令来控制或与 Docker 守护程序交互。许多其他 Docker 应用程序也使用底层的 API 和 CLI。

守护进程创建和管理 Docker 对象，如镜像（image）、容器、网络和数据卷。

2. Docker 的架构

Docker 使用客户端—服务器模式的架构，如图 8-5 所示。Docker 客户端可以与 Docker 守护进程通信，而 Docker 守护进程会构建、运行和分发 Docker 容器。Docker 客户端和守护程序可以在同一系统上运行，也可以将 Docker 客户端连接到远程 Docker 守护程序。Docker 客户端和守护程序通过 socket 或 REST API 进行通信。

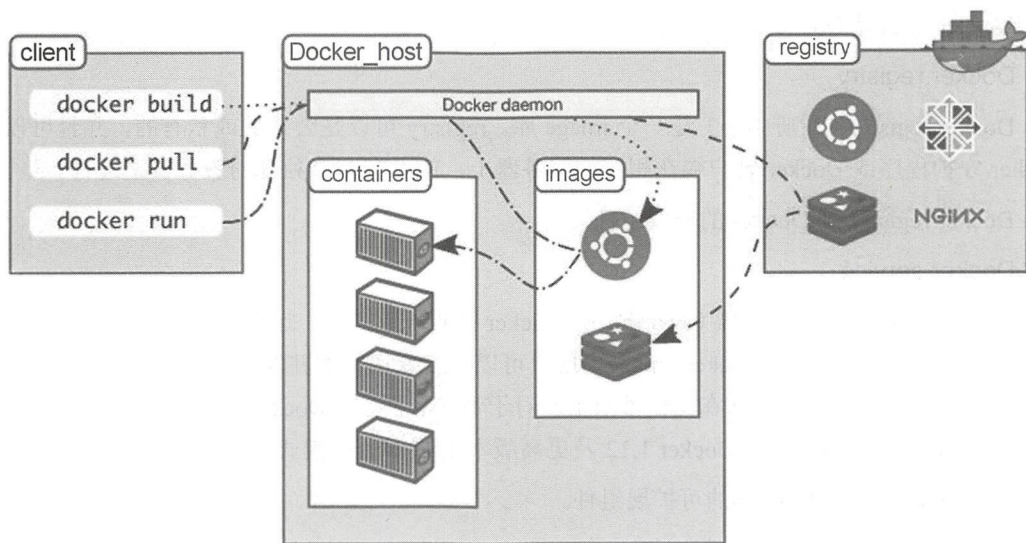


图 8-5 Docker 架构

Docker 守护进程

Docker 守护程序在主机上运行。用户使用 Docker 客户端与守护程序进行交互。

Docker 客户端

Docker 客户端是 Docker 二进制数形式的，是 Docker 的主要用户接口。它从用户那里接收命令和配置标识，并与 Docker 守护程序进行通信。一个客户端甚至可以与多个不相关的守护程序通信。

Docker image

Docker image（镜像）是一个只读模板，包含用于创建 Docker 容器的说明。例如，image 可能包含带有 Apache Web 服务器和 Web 应用程序的 Ubuntu 操作系统。用户可以从头开始构建或更新 image，也可以下载并使用其他人创建的 image。image 可以基于和扩展一个或多个其他的 image。Docker image 在文本文件中被称为 Dockerfile，它有一个简单的、定义良好的语法。

Docker image 是 Docker 的构建组件。

Docker 容器

Docker 容器（container）是 Docker image 的可运行实例。用户可以使用 Docker API 或 CLI 命令运行、启动、停止、移动或删除容器。运行容器时，可以提供配置元数据，例如，网络信息或环境变量。每个容器是一个隔离、安全的应用程序平台，但可以访问在不同主机或容器中运行的资源。



Docker 容器是 Docker 的运行组件。

Docker registry

Docker registry（注册中心）是一个 image 库。registry 可以是公共的或私有的，并且可以与 Docker 守护程序或 Docker 客户端在相同的服务器上，或者在完全独立的服务器上运行。

Docker registry 是 Docker 的分发组件。

Docker service

Docker service 允许一个群（swarm）的 Docker 节点一起工作，运行一个预定数量的副本任务的实例，其本身是一个 Docker image。用户可以指定要运行的并发副本任务的数量，swarm 管理器会确保将负载均匀分布在工作节点上。对于消费者而言，Docker 服务就像一个单一的应用程序。Docker Engine 支持 Docker 1.12 及更高版本中的 swarm 模式。

Docker service 是 Docker 的可扩展组件。

3. Docker image 的工作原理

Docker image 是只读模板，从 Docker 容器中实例化而来。每个 image 由一系列层组成。Docker 使用 union file systems（联合文件系统）将这些图层组合成单个 image。union file system 允许单独文件系统的文件和目录（称为分支）被透明地覆盖，形成单个一致的文件系统。

这些层是 Docker 如此轻量级的原因之一。当更改 Docker image 时，例如，将应用程序更新到新版本，将创建一个新层，并仅替换其要更新的层，其他层原封不动。要分发更新，只需传输要更新的层即可。分层加速了 Docker 镜像的分发。Docker 会确定哪些层需要在运行时更新。

image 在 Dockerfile 中定义。每个 image 从基本 image（例如，ubuntu，一个基本 Ubuntu image；或 fedora，一个基本 Fedora image）开始。用户还可以使用自己的 image 作为新 image 的基础，例如，如果用户有一个基本 Apache image，则可以将其作为所有 Web 应用程序 image 的基础。基本 image 是使用 dockerfile 中的 FROM 关键字来定义的。

注：Docker Hub 是一个公共的 registry，用于存储 image。

image 是使用我们称为 instruction（指令）的简单的描述性步骤从基本 image 来构建的，这些步骤存储在 Dockerfile 中。每个 instruction 在 image 中创建一个新层。下面是 Dockerfile instruction 的一些示例：

- 指定基本 image（FROM）；
- 指定维护者（MAINTAINER）；
- 运行命令（RUN）；
- 添加文件或目录（ADD）；



- 创建环境变量（ENV）；
- 当容器从 image 启动时决定运行哪个进程（CMD）。

当请求构建映像，执行 instruction 并返回 image 时，Docker 会读取此 Dockerfile。

4. Docker registry 的工作原理

Docker registry 用于存储 Docker image。在构建 Docker image 之后，可以将其推送到公共 registry（如 Docker Hub）或运行在防火墙后面的私有 registry 中。还可以搜索现有 image，并将其从 registry 中拉到主机中。

Docker Hub 是一个公共的 Docker registry 表，它提供一个巨大的现有 image 的集合，并允许用户贡献自己的 image 到该 registry 中。

Docker Store 允许用户购买和销售 Docker image。对于 image，用户可以从软件供应商处购买包含应用程序或服务的 Docker image，然后使用该 image 将应用程序部署到测试、暂存和生产环境中，通过提取新版本的 image 来升级应用程序，并重新部署容器。截至本书出版时，Docker Store 处于私有测试阶段。

5. 容器的工作原理

容器使用主机的 Linux 内核，包含在创建 image 时添加的所有额外文件，以及在创建或启动容器时与容器关联的元数据。每个容器由 image 构建。该 image 定义容器的内容，包括启动容器时运行的进程及各种其他配置的详细信息。Docker 镜像是只读的。当 Docker 从 image 运行容器时，它会在应用程序运行的 image 的顶部（使用我们之前看到的 union file systems）添加一个读写层。

当使用 `docker run` CLI 命令或等效的 API 时，Docker Engine 客户端指示 Docker 守护程序运行容器。下面的示例告诉 Docker 守护程序使用 `ubuntu` Docker image 来运行容器，以交互模式（-i）保持在前台，并运行 `/bin/bash` 命令。

```
$ docker run -i -t ubuntu /bin/bash
```

当运行此命令时，Docker Engine 执行以下操作。

- 拉取 Ubuntu image：Docker Engine 检查 Ubuntu image 的存在。如果 image 已在本地存在，则 Docker Engine 会将其用于新容器。否则，Docker Engine 会从 Docker Hub 中获取它。
- 创建一个新容器：Docker 使用 image 创建一个容器。
- 分配文件系统并装载读写层：容器在文件系统中创建，并将读写层添加到 image 中。
- 分配网络/网桥接口：创建一个允许 Docker 容器与本地主机通信的网络接口。



- 设置 IP 地址：从池中查找并附加可用的 IP 地址。
- 执行指定的进程：执行/bin/bash 可执行文件。
- 捕获并提供应用程序输出：在交互模式下，建立所有连接后，会记录标准输入、输出和错误，以便查看应用程序的运行状况。

6. Docker 底层技术

Docker 利用 Linux 内核的几个功能来支撑它的功能。

命名空间

Docker 使用一种称为命名空间（namespaces）的技术来提供容器的隔离工作空间。当运行一个容器时，Docker 为该容器创建一组命名空间。

这些命名空间提供一个隔离层。容器的每个方面都运行在单独的命名空间中，并且其访问仅限于该命名空间。

Docker Engine 在 Linux 上使用如下命名空间。

- pid 命名空间——进程隔离（PID：进程 ID）；
- net 命名空间——管理网络接口（NET：网络）；
- ipc 命名空间——管理对 IPC 资源的访问（IPC：进程间通信）；
- mnt 命名空间——管理文件系统安装点（MNT：Mount）；
- uts 命名空间——隔离内核和版本标识符（UTS：UNIX 分时系统）。

控制组

Linux 上的 Docker Engine 还依赖于另一种称为控制组（control groups，简称 cgroups）的技术。控制组将应用程序限制为特定的一组资源。控制组允许 Docker Engine 将可用的硬件资源共享到容器中，并且可选地实施限制和约束。例如，用户可以限制特定容器可用的内存。

联合文件系统

联合文件系统（Union file systems，简称 UnionFS）是通过创建层来操作的文件系统，使它们非常轻量 and 快速。Docker Engine 使用 UnionFS 提供容器的构建块。Docker Engine 可以使用多个 UnionFS 变体，包括 AUFS、btrfs、vfs 和 DeviceMapper。

容器格式

Docker Engine 将命名空间、控制组和 UnionFS 组合成一个称为容器格式的包装器。默认的容器格式为 libcontainer。将来 Docker 可以通过与诸如 BSD Jails 或 Solaris Zones 等技术集成来支持其他容器格式。





8.3.3 Docker 的使用

下面对 Docker 的使用进行简单的介绍。

1. 运行 Hello world 程序

运行一个 Hello world 容器：

```
$ docker run ubuntu /bin/echo 'Hello world'
Hello world
```

这样就启动了容器。

在该例子中：

- `docker run` 是运行容器的指令；
- `ubuntu` 是要运行的 Ubuntu 操作系统的 image；
- `/bin/echo` 是运行内部新容器的指令。

当容器启动后，Docker 会创建一个新的 Ubuntu 环境，并执行 `/bin/echo` 命令，接着输出如下：

```
Hello world
```

2. 运行一个可交互的容器

执行如下命令：

```
$ docker run -t -i ubuntu /bin/bash
root@af8bae53bdd3:/#
```

在该例子中：

- `docker run` 是运行容器的指令；
- `ubuntu` 是要运行的 Ubuntu 操作系统的 image；
- `-t` 标识在新容器内分配虚拟终端或终端；
- `-i` 通过抓取容器的（STDIN）中的标准来进行交互式连接；
- `/bin/echo` 是运行内部新容器的指令。

这样就可以在容器内运行一些指令：

```
root@af8bae53bdd3:/# pwd
/
root@af8bae53bdd3:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp
usr var
```





3. 以守护进程方式启动 Hello world

使用以下方式创建并启动一个守护进程：

```
$ docker run -d ubuntu /bin/sh -c "while true; do echo hello world; sleep 1; done"
```

```
1e5535038e285177d5214659a068137486f96ee5c2e85a4ac52dc83f2ebe4147
```

在该例子中：

- `docker run` 是运行容器的指令；
- `-d` 标识以后台方式运行容器；
- `ubuntu` 是要运行的 Ubuntu 操作系统的 image。

最后，指定以下方式运行：

```
/bin/sh -c "while true; do echo hello world; sleep 1; done"
```

输出如下：

```
1e5535038e285177d5214659a068137486f96ee5c2e85a4ac52dc83f2ebe4147
```

这个字符串就是容器 ID。

以下方式可以查看容器的运行状态：

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
1e5535038e28	ubuntu	/bin/sh -c 'while tr	2 minutes ago	Up	1 minute	insane_babbage

其中：

- `1e5535038e28` 是容器 ID 的简写变体；
- `ubuntu` 是运行的 Ubuntu 操作系统的 image；
- 命令、状态和分配的名称是 `insane_babbage`。

执行：

```
$ docker logs insane_babbage
hello world
hello world
hello world
. . .
```

其中，`docker logs` 用于查看容器的内容信息，并返回 `hello world`。





停止容器，执行：

```
$ docker stop insane_babbage
insane_babbage
```

再次查看容器的状态：

```
$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS   NAMES
```

可以看到容器已经停止了。

8.4 实战：基于 Docker 构建、运行、发布微服务

Docker 可以说是目前市面上最火爆的容器技术之一。本节我们将使用 Docker 来演示如何构建、运行、发布微服务。

8.4.1 编写微服务

为了方便演示，我们采用 Spring Boot 编写一个微服务应用 hello-world-docke。该应用非常简单，内容如下：

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController

public class HelloController {

    @RequestMapping("/hello")
    public String hello() {
        return "Hello World! Welcome to visit waylau.com!";
    }
}
```

同时，我们执行 gradlew build 来编译 hello-world-docker 应用。编译成功之后，就能运行该编译文件：

```
java -jar build/libs/hello-world-docker-1.0.0.jar
```

此时，在浏览器中访问 <http://localhost:8080/hello> 应能看到 “Hello World! Welcome to visit waylau.com!” 字样的内容，说明该微服务构建成功。





8.4.2 微服务容器化

我们需要将微服务应用包装为 Docker 容器。Docker 使用 Dockerfile 文件格式来指定 image 层。

我们在 hello-world-docke 应用的根目录下创建 Dockerfile 文件：

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE
ADD ${JAR_FILE} app.jar
ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar",
"/app.jar"]
```

这个 Dockerfile 是非常简单的，因为本例子中的微服务应用相对比较简单。其中：

- FROM 可以理解为这个 image 依赖于另外一个 image。因为我们的应用是一个 Java 应用，所以依赖于 JDK。
- 项目 JAR 文件以“app.jar”的形式添加到容器中，然后在 ENTRYPOINT 中执行。
- VOLUME 指定临时文件目录为/tmp。其效果是在主机/var/lib/docker 目录下创建一个临时文件，并链接到容器的/tmp。该步骤是可选的，如果涉及文件系统的应用就很有必要了。/tmp 目录用来持久化到 Docker 数据文件夹，因为 Spring Boot 内嵌的 Tomcat 容器默认使用/tmp 作为工作目录。
- 为了缩短 Tomcat 的启动时间，添加一个系统属性指向/dev/./urandom。

8.4.3 构建 Docker image

为了使用 Gradle 来构建 Docker image，需要在应用的 build.gradle 中添加 docker 插件。

```
buildscript {
    ...
    dependencies {
        ...
        classpath('gradle.plugin.com.palantir.gradle.docker:gradle-
docker:0.17.2')
    }
}
...
```





```

apply plugin: 'com.palantir.docker'

docker {
    name "${project.group}/${jar.baseName}"
    files jar.archivePath
    buildArgs(['JAR_FILE': "${jar.archiveName}"])
}

```

执行 `gradlew build docker` 来构建 Docker image:

```
> gradlew build docker --info
```

```
...
```

```
Starting process 'command 'docker''. Working directory:
```

```

D:\workspaceGitosc\spring-cloud-microservices-development\samples\hello-world-docker\build\docker Command: docker build --build-arg
JAR_FILE=hello-world-docker-1.0.0.jar -t
com.waylau.spring.cloud/hello-world-docker .

```

```
Successfully started process 'command 'docker''
```

```
Sending build context to Docker daemon 15.18MB
```

```
Step 1/5 : FROM openjdk:8-jdk-alpine
```

```
8-jdk-alpine: Pulling from library/openjdk
```

```
2fdfe1cd78c2: Pulling fs layer
```

```
82630fd6e5ba: Pulling fs layer
```

```
001511eb3437: Pulling fs layer
```

```
82630fd6e5ba: Verifying Checksum
```

```
82630fd6e5ba: Download complete
```

```
2fdfe1cd78c2: Verifying Checksum
```

```
2fdfe1cd78c2: Download complete
```

```
2fdfe1cd78c2: Pull complete
```

```
82630fd6e5ba: Pull complete
```

```
001511eb3437: Verifying Checksum
```

```
001511eb3437: Download complete
```

```
001511eb3437: Pull complete
```

```
Digest: sha256:388566cc682f59a0019004c2d343dd6c69b83914dc5c458be959271af2761795
```

```
Status: Downloaded newer image for openjdk:8-jdk-alpine
```

```
---> 3642e636096d
```

```
Step 2/5 : VOLUME /tmp
```





```
---> Running in 40ff6fa809e8
---> f467a7dlc267
Removing intermediate container 40ff6fa809e8
Step 3/5 : ARG JAR_FILE
---> Running in 4872c7353093
---> 4406b96eca35
Removing intermediate container 4872c7353093
Step 4/5 : ADD ${JAR_FILE} app.jar
---> a2e55472f1db
Step 5/5 : ENTRYPOINT java -Djava.security.egd=file:/dev/./urandom -jar
/app.jar
---> Running in f536a4993ca5
---> 527b7c667dd2
Removing intermediate container f536a4993ca5
Successfully built 527b7c667dd2
Successfully tagged com.waylau.spring.cloud/hello-world-docker:latest
SECURITY WARNING: You are building a Docker image from Windows against a
non-Windows Docker host. All files and directories added to build context will
have '-rwxr-xr-x' permissions. It is recommended to double check and reset
permissions for sensitive files and directories.

:docker (Thread[Task worker,5,main]) completed. Took 15 mins 24.218 secs.

BUILD SUCCESSFUL in 15m 26s
9 actionable tasks: 3 executed, 6 up-to-date
Stopped 0 worker daemon(s).
```

构建成功，可以在控制台看到如上信息。因篇幅有限，省去大部分内容。

8.4.4 运行 image

在构建 Docker image 完成之后使用 Docker 来运行该 image:

```
docker run -p 8080:8080 -t com.waylau.spring.cloud/hello-world-docker
```

图 8-6 展示了运行 image 的过程。




```
命令提示符 - docker run -p 8080:8080 -t com.waylau.spring.cloud/hello-world-docker
C:\Users\Administrator>docker run -p 8080:8080 -t com.waylau.spring.cloud/hello-world-docker

Spring
=====
:: Spring Boot ::
               (v2.0.0.M4)

2017-12-23 14:29:10.745 INFO --- [main] c.w.spring.cloud.weather.Application : Starting Application on
7dde27af2b7b with PID 1 (/app.jar started by root in /)
2017-12-23 14:29:10.761 INFO --- [main] c.w.spring.cloud.weather.Application : No active profile set,
falling back to default profiles: default
2017-12-23 14:29:11.004 INFO --- [main] ConfigServletWebServerApplicationContext : Refreshing org.springframework
amework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@18e3568: startup date [Sat Dec 23 14
:29:10 GMT 2017]; root of context hierarchy
2017-12-23 14:29:14.972 INFO --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with
port(s): 8080 (http)
2017-12-23 14:29:15.020 INFO --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2017-12-23 14:29:15.030 INFO --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine
: Apache Tomcat/8.5.20
2017-12-23 14:29:15.442 INFO --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring emb
edded WebApplicationContext
2017-12-23 14:29:15.442 INFO --- [ost-startStop-1] o.s.web.context.ContextLoader : Root WebApplicationCont
ext: initialization completed in 4465 ms
2017-12-23 14:29:15.661 INFO --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean : Mapping servlet: 'dispa
tcherServlet' to [/]
2017-12-23 14:29:15.691 INFO --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'charac
```

图 8-6 运行 image 的过程

8.4.5 访问应用

image:image 运行成功之后，就能在浏览器中访问 <http://localhost:8080/hello>，应该能看到“Hello World! Welcome to visit waylau.com!”。

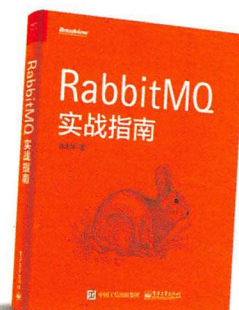
8.4.6 发布微服务

当我们的微服务包装成 Docker 的 image 之后，就能进行分发了。Docker Hub 是专门用于托管 image 的云服务。用户可以将自己的 image 推送到 Docker Hub 上，以方便其他人下载。

上述代码可以在 <https://github.com/waylau/distributed-systems-technologies-and-cases-analysis> 的 hello-world-docker 程序中找到。



好书分享



拒绝堆砌臃肿，支持纯正原创

投稿: chenxm@phei.com.cn





分布式系统 常用技术及案例分析

(第2版)

读者服务

轻松注册成为博文视点社区用户
(www.broadview.com.cn),
扫码直达本书页面。

★**提交勘误**: 您对书中内容的修改意见可在 提交勘误处提交, 若被采纳, 将获赠博文视点社区积分(在您购买电子书时, 积分可用来抵扣相应金额)。

★**交流互动**: 在页面下方 读者评论 处留下您的疑问或观点, 与我们和其他读者一同学习交流。

页面入口: <http://www.broadview.com.cn/35677>



博文视点Broadview



@博文视点Broadview



责任编辑: 陈晓猛
封面设计: 李 玲

上架建议: 计算机 / 分布式系

ISBN 978-7-121-35677-3



定价: 99.00元